ANDRÉ MURBACH MAIDL

UMA IMPLEMENTAÇÃO DA SEMÂNTICA DE AÇÕES ORIENTADA A OBJETOS EM MAUDE

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Martin A. Musicante
Co-orientador: André Luiz Pires Guedes

CURITIBA

2007

AGRADECIMENTOS

A Deus e todas as energias que regem o universo.

Ao meu pai José Carlos e à minha mãe Ariete, pela preocupação que têm com a minha educação, por sempre incentivarem o meu interesse pela computação e também por terem me proporcionado o primeiro contato com a área.

À minha noiva Izabella, por ser a minha fonte de inspiração e por todo o apoio durante o desenvolvimento deste trabalho.

Ao amigo e professor Cláudio Carvilhe, por me apresentar a Semântica de Ações, além do seu grande interesse e ajuda neste trabalho.

Ao professor Martin A. Musicante, pela confiança em mim creditada durante todo o trabalho e pelas suas orientações.

Ao professor André Luiz Pires Guedes, pelo seu trabalho de co-orientador.

Ao amigo André Hochuli, por me alertar sobre o processo de inscrição da Universidade Federal do Paraná e também por finalizá-lo para mim.

Aos amigos: Adriano Del Vigna, Alexandre Cavedon, Davi Arnaut, Diego Antunes, Diogo Salgueiro, Felipe Pardal e Luis Cláudio, da extinta equipe de desenvolvimento Linux FADT, pela compreensão nos momentos em que estive ausente.

Aos desenvolvedores da MMT, Christiano Braga e Fabricio Chalub, pelo interesse e toda a ajuda prestada ao usar a ferramenta mencionada para desenvolver este trabalho.

Aos professores Roberto Bigonha e Silvia Vergilio, pelos seus comentários construtivos em relação à esta dissertação.

À Universidade Federal do Paraná pela oportunidade concedida e pelo apoio na divulgação de parte deste trabalho.

SUMÁRIO

LISTA DE FIGURAS

RESUMO

Nesta dissertação apresentamos Maude Object-Oriented Action Tool, um ambiente executável para a Semântica de Ações Orientada a Objetos, escrito como uma extensão conservativa de Full Maude e Maude MSOS Tool. A Notação de Ações foi implementada usando as transições MSOS disponibilizadas por Maude MSOS Tool, enquanto a Notação de Classes foi implementada usando Full Maude. A sintaxe criada pela ferramenta é bastante similar a do formalismo. Além da ferramenta, também apresentamos a Language Features Library versão 2, uma nova versão da biblioteca de classes para a Semântica de Ações Orientada a Objetos. Esta nova versão resolve alguns dos problemas encontrados na versão anterior, basicamente pelo uso de abstrações de ações em vez de passagem de parâmetros entre classes. Além disso, mostramos a Semântica de Ações Orientada a Objetos Construtiva, uma combinação entre a Semântica de Ações Orientada a Objetos e a Semântica de Ações Construtiva. Esta é a primeira implementação da Semântica de Ações Orientada a Objetos, a qual usamos para testar e comparar o uso da biblioteca e da abordagem construtiva no formalismo orientado a objetos.

ABSTRACT

In this dissertation we present Maude Object-Oriented Action Tool, an executable environment for Object-Oriented Action Semantics implemented as a conservative extension of Full Maude and Maude MSOS Tool. The Modular SOS of Action Notation has been implemented using MSOS transitions provided by Maude MSOS Tool and Full Maude has been used to implement the Classes Notation. The syntax created by the tool is fairly similar to the formalism one. Furthermore, we also present Language Features Library version 2, a new version for the Object-Oriented Action Semantics library of classes. The new library solves some problems found on the previous one, basically by using abstraction of actions instead of passing parameters among classes. Moreover, we show how Constructive Object-Oriented Action Semantics might be achieved as a combination between Object-Oriented Action Semantics and Constructive Action Semantics. This is the first implementation of Object-Oriented Action Semantics, which has been used to test and compare the use of the library and the constructive approach in the object-oriented formalism.

CAPÍTULO 1

INTRODUÇÃO

A especificação formal de linguagens de programação é uma etapa crucial no desenvolvimento de uma determinada linguagem. É nessa etapa que as características específicas da linguagem são expressas matematicamente, sendo assim é possível inibir certos comportamentos indesejados, evitar ambigüidade nas frases da linguagem e proporcionar a base para a sua implementação propriamente dita. Utilizar um método formal para definir uma linguagem de programação permite que ela seja melhor conhecida pelos seus projetistas, implementadores e usuários.

Dentre os formalismos existentes, para a especificação formal de linguagens de programação, destacamos a Semântica Denotacional [Wat91, Sch86], a Semântica Operacional [Plo81, Win93] e a Semântica de Ações [Mos92].

A Semântica Denotacional foi uma das primeiras técnicas usadas para a especificação formal de linguagens de programação. Ela utiliza conceitos puramente matemáticos e por isso no início do seu desenvolvimento foi chamada de "Semântica Matemática". Por estar fortemente atrelada à teoria matemática permite um rico detalhamento da especificação, contudo essas especificações acabam tornando-se complexas, dificultando a compreensão, modificação e extensão.

A Semântica de Ações é baseada na Semântica Denotacional e expressa o significado de um programa por meio de uma entidade matemática chamada ação. Este formalismo disponibiliza uma notação formal, que faz uso de palavras da língua inglesa, para ser usada na descrição de linguagens. Dessa forma facilita a compreensão, modificação e extensão de uma determinada especificação. Porém, este formalismo não proporciona a base necessária para a criação de bibliotecas [Gay02].

Para suprir a necessidade de criação de bibliotecas, algumas novas técnicas foram propostas como uma extensão para a Semântica de Ações. A Semântica de Ações baseada

em módulos [DM01] utiliza a combinação de módulos separados, já a Semântica de Ações baseada em componentes [MM01] compõem diferentes componentes e a Semântica de Ações Orientada a Objetos [Car02, CM03] faz uso de conceitos da orientação a objetos, como por exemplo definições de classes e métodos. As três abordagens solucionam alguns dos problemas de reutilização de código encontrados na abordagem original.

Apesar de alguns problemas terem sido resolvidos, um novo foi inserido. O reaproveitamento de código, seja ele classe, módulo ou componente, é de fato a cópia de um trecho de código já existente no projeto atual. Esse problema foi resolvido na Semântica de Ações Orientada a Objetos pela criação de uma biblioteca de classes, a LFL [Ara04, AM04]. Ao usar essa biblioteca, pode-se incluir diversas características presentes nas linguagens e paradigmas mais comuns apenas ao mencionar uma das classes por ela disponibilizada.

1.1 Motivação

Cada linguagem de programação possui características diferentes uma das outras. Mas, elas também costumam possuir características em comum. Além disso, algumas características estão presentes em várias linguagens, como por exemplo: declarações locais, iterações, amarrações, condições e funções abstratas.

Em alguns casos devemos considerar alguns detalhes da forma e da interpretação pretendida de cada característica da linguagem, como por exemplo: a forma como expressões de condição são escritas, e se a condição é numérica ou booleana. No entanto, se nós transformarmos uma determinada característica em um construtor abstrato individual com a devida interpretação pretendida para ele, então essa característica pode ser freqüentemente encontrada em um número significante de linguagens de programação.

Normalmente, a descrição semântica de uma determinada característica é expressa somente em descrições de uma linguagem completa e características comuns acabam sendo reescritas várias vezes. Na maioria dos métodos para a especificação da semântica de linguagens, a descrição de cada característica depende criticamente de quais outros tipos de características são inclusos na linguagem descrita.

Em [Mos05] é proposta uma abordagem construtiva, a qual suporta essas definições

independentes e usa nome de módulos para descrever características individuais de linguagens. Tal abordagem tem sido usada com a Semântica de Ações de forma a obter a Semântica de Ações Construtiva [Ive05]. Acredita-se que a biblioteca de classes disponibilizada pela LFL possui características semelhantes a abordagem construtiva. Ambas as abordagens introduziram um conceito interessante na definição formal de linguagens: a descrição semântica de uma linguagem independentemente da sintaxe dela.

No entanto, a forma com que a LFL foi definida introduziu alguns problemas de sintaxe na definição de linguagens usando a Semântica de Ações Orientada a Objetos. Para ser mais exato, o mecanismo de passagem de parâmetros é um tanto obscuro, além de voltar os problemas de leitura e escrita encontrados em outras descrições semânticas de linguagens de programação.

Não obstante, como reportado em [vdBIM06], nenhuma ferramenta para a definição formal de linguagens usando a Semântica de Ações Orientada a Objetos tinha sido disponibilizada. Conseqüentemente, até o presente momento, a biblioteca de classes também não tinha sido implementada e testada.

1.2 Objetivos

A Semântica de Ações Orientada a Objetos foi definida usando transições da Semântica Operacional Estrutural [Plo81] em [Car02]. Nesta dissertação apresentaremos MOOAT (*Maude Object-Oriented Action Tool*). A primeira ferramenta para descrever linguagens de programação usando a Semântica de Ações Orientada a Objetos.

Algumas mudanças na sintaxe e semântica da Semântica de Ações Orientada a Objetos são propostas pela implementação. Uma diferença importante entre a ferramenta e o formalismo é que MOOAT usa Semântica Operacional Estrutural Modular [Mos04] em vez de usar apenas a Semântica Operacional Estrutural.

Tal ferramenta foi desenvolvida em Maude [CDE+05]. A Notação de Classes, proposta em [Car02], foi desenvolvida usando operações e equações de Full Maude [Dur99, DM99], enquanto a Notação de Ações foi especificada usando transições da Semântica Operacional Estrutural Modular implementada pela MMT (*Maude MSOS Tool*), tendo como base as

transições e módulos descritos em [Mos99].

Além da implementação, propomos também a LFLv2 (LFL versão 2), uma nova versão da LFL para resolver alguns problemas da versão original da biblioteca. Nela, incorporamos algumas das idéias apresentadas em [Mos05, Ive05], com o objetivo de separar a sintaxe da linguagem de programação da especificação da sua semântica.

Alguns estudos de caso foram implementados para testar efetivamente a implementação de MOOAT. O primeiro deles é a especificação de uma linguagem imperativa simples, com o intuito de apenas demonstrar a ferramenta. Um outro estudo de caso é a especificação e teste da LFLv2. Ainda outro é a Semântica de Ações Orientada a Objetos Construtiva, a qual é basicamente uma combinação entre a Semântica de Ações Orientada a Objetos e a Semântica de Ações Construtiva, com o objetivo de comparar a abordagem construtiva e a biblioteca.

1.3 Estrutura desta dissertação

O trabalho está organizado da seguinte forma: uma visão geral da Semântica de Ações e de algumas das suas variações são apresentadas no Capítulo 2. As ferramentas usadas no desenvolvimento de MOOAT são brevemente descritas no Capítulo 3. No Capítulo 4 é proposta uma nova versão para a biblioteca de classes. A notação de MOOAT, bem como a sua implementação são descritas no Capítulo 5. No Capítulo 6 são apresentados alguns casos de estudo usando a ferramenta descrita. As conclusões e trabalhos futuros são apresentados no Capítulo 7.

CAPÍTULO 2

USANDO A SEMÂNTICA DE AÇÕES

Neste capítulo será exposta uma breve introdução à semântica formal de linguagens de programação, necessária para a compreensão da *Semântica de Ações* [Mos92]. Em seguida estudaremos o formalismo da semântica de ações propriamente dito e algumas das suas variações.

2.1 Semântica formal de linguagens de programação

Linguagem é um conjunto de caracteres, símbolos e regras utilizado para dar instruções a um computador. Ela é um artefato e deve ser desenvolvida concisamente. Mais ainda, uma linguagem de programação deve ser simples e direta, pois como será usada para dar instruções a máquinas não pode conter ambigüidade [Wat91].

A semântica de uma linguagem expressa o significado das suas frases. Por meio dela descreve-se o comportamento de comandos, declarações, expressões e outras estruturas da linguagem. A necessidade da especificação formal de linguagens revela as ambiguidades da linguagem e é uma base para a sua implementação.

Veremos nesta seção duas abordagens capazes de descrever a semântica formal de linguagens de programação, são elas: *Semântica Denotacional e Semântica Operacional.*

2.1.1 Sintaxe de linguagens de programação

A sintaxe de uma linguagem é um conjunto de regras, o qual define a forma das frases da linguagem. Esse conjunto de regras também é chamado de gramática da linguagem, a qual serve de base à elaboração de programas que reconhecem estas frases. A sintaxe não diz respeito ao significado da sentença, isso fica a cargo da semântica, ou seja, a sintaxe apenas descreve a maneira como as sentenças são escritas e não o seu significado.

Existem duas abordagens para a especificação da sintaxe de uma linguagem. Uma

delas é a *sintaxe abstrata*, a qual permite descrever a linguagem usando menos regras, porém de melhor compreensão para o leitor e a outra é a *sintaxe concreta*, a qual segue à risca as regras da linguagem, ou seja, ela define todas as regras que são ignoradas na primeira abordagem citada.

Uma notação concisa deve ser usada para especificar a sintaxe de uma linguagem. Por isso, expressaremos ambas as abordagens, abstrata e concreta, por meio de gramáticas livres de contexto na notação **Backus-Naur (BNF)** [ASU88].

2.1.1.1 Sintaxe abstrata

A sintaxe abstrata se preocupa apenas com a relação hierárquica entre as frases de uma linguagem, por exemplo: um **while-command** consiste de uma expressão e um comando. Na abordagem abstrata a ambigüidade não é levada em consideração porque nela uma linguagem é definida com o intuito de expressar a sua estrutura usando uma descrição simples e de alto nível. Segue abaixo a definição da sintaxe abstrata de uma linguagem de expressões aritméticas.

Exemplo 2.1.1 (Sintaxe abstrata de uma linguagem de expressões) *:*

$$a \quad ::= \quad n \mid a + a \mid a - a \mid a * a \tag{2.1}$$

A regra 2.1 define de forma abstrata a sintaxe de uma linguagem de expressões aritméticas. O elemento sintático n representa um valor inteiro. O elemento sintático a representa o conjunto das expressões aritméticas.

A sintaxe abstrata ignora o "*parsing*" da sintaxe, nenhuma regra de precedência de operadores foi definida. No caso desse exemplo ela se preocupa apenas com a estrutura das expressões e com seus operadores.

2.1.1.2 Sintaxe concreta

Diferente da abordagem anterior, a sintaxe concreta se preocupa com a gramática completa a ser utilizada para analisar sintaticamente a linguagem. É nela que definimos, por exemplo: precedência de operadores e tratamento de parênteses em expressões. A seguir é definida a sintaxe concreta da linguagem de expressões aritméticas do Exemplo 2.1.1.

Exemplo 2.1.2 (Sintaxe concreta de uma linguagem de expressões) *:*

$$a \ ::= \ t + a \mid \ t - a \mid \ t \tag{2.2}$$

$$t \ ::= \ f * t \mid \ f \tag{2.3}$$

$$f \ ::= \ n \tag{2.4}$$

Na sintaxe concreta novas estruturas apareceram, são elas: t e f. Estamos usando símbolos não-terminais (a, t e f) para definir a linguagem. O símbolo a representa a expressão em si enquanto t e f representam respectivamente termos e fatores.

Essa divisão caracteriza a precedência e a associatividade dos operadores, o que possibilita criar uma única árvore de "*parsing*" para cada expressão aritmética, sendo assim eliminamos a possibilidade de uma escrita ambígua, a qual não era evitada pela sintaxe abstrata.

Examinado o exemplo acima percebe-se que a regra 2.2 define a possibilidade da soma ($a + t$) e/ou subtração ($a - t$) de uma expressão com um termo e também que uma expressão pode ser apenas um termo (t). Em 2.3 o conceito de termo é definido como a multiplicação dele mesmo por um fator ($t * f$) ou apenas o próprio fator (f). Finalmente, 2.4 representa um fator como um número inteiro já avaliado (n). Nota-se claramente que a multiplicação deve ser avaliada, obrigatoriamente, antes de uma soma ou de uma subtração, pois os fatores estão agrupados com os termos enquanto os termos estão agrupados com as expressões e os fatores são mais restritos que os termos que são mais restritos que as expressões.

A *sintaxe concreta* é muito útil na criação de **interpretadores** e/ou **compiladores**, por ser mais detalhada. No caso de apenas expressar o significado de uma linguagem a *sintaxe abstrata* é mais interessante, por ser mais independente dos detalhes do "*parsing*" das frases da linguagem.

2.1.2 Semântica Denotacional

A Semântica Denotacional é um método utilizado para a especificação da semântica de linguagens de programação. Foi desenvolvida por Christopher Strachey e Dana Scott nos anos 70 e tem uma base puramente matemática, por isso quase foi chamada de *Semântica Matemática* [Wat91]. A denominação denotacional foi adotada, porque o significado de cada frase de um programa é chamado de *denotação*. Para o nosso trabalho é importante esclarecermos o significado dos seguintes conceitos da Semântica Denotacional: *funções semânticas*, *equações semânticas*, *armazenamento* e *ambiente de avaliação*. Esta seção apresentada está baseada em [Car02, Wat91].

2.1.2.1 Funções semânticas e equações semânticas

Na Semântica Denotacional representamos o significado de cada frase por meio de uma entidade matemática, a qual é chamada de *denotação*. A semântica de uma linguagem é especificada por meio de funções que mapeiam essas frases para as suas denotações correspondentes, as quais chamamos de *funções semânticas*.

Vejamos as definições de função semântica e equação semântica de acordo com [Wat91]:

- *função semântica* - $f: P \rightarrow D$,
  onde f é uma função que mapeia uma determinada frase P para a sua respectiva denotação D.

- *equação semântica* - $f\ [\![\ldots Q \ldots R \ldots]\!] = \ldots f'Q \ldots f''R \ldots$,
  onde para cada função semântica *f*, definida, há uma equação semântica correspondente. As subfrases Q e R formam a frase P de tal forma que f' e f'' são funções semânticas definidas para Q e R respectivamente.

A seguir, exemplificaremos a Semântica Denotacional das frases definidas pela sintaxe abstrata do Exemplo 2.1.1, usando os conceitos de funções e equações semânticas:

Exemplo 2.1.3 (Semântica Denotacional de uma linguagem de expressões) *:*

- evaluate : Expression $\rightarrow$ Integer
- valuation : Number $\rightarrow$ Integer
- sum : Integer $\times$ Integer $\rightarrow$ Integer
- difference : Integer $\times$ Integer $\rightarrow$ Integer
- product : Integer $\times$ Integer $\rightarrow$ Integer

(1) evaluate $[\![$ N:Number $]\!]$ = valuation N

(2) evaluate $[\![$ E_1:Expression "+" E_2:Expression $]\!]$ = sum(evaluate E_1, evaluate E_2)

(3) evaluate $[\![$ E_1:Expression "-" E_2:Expression $]\!]$ = difference(evaluate E_1, evaluate E_2)

(4) evaluate $[\![$ E_1:Expression "*" E_2:Expression $]\!]$ = product(evaluate E_1, evaluate E_2)

Os termos Expression e Number representam, respectivamente, os elementos sintáticos a e n. Foram usadas as variáveis E e N para facilitar o uso dos termos Expression e Number.

A *função semântica* evaluate mapeia uma expressão, denotada por Expression, para um número inteiro, o qual faz parte do sort Integer. Por sua vez, valuation representa a avaliação de um número, denotado por Number, para um valor do sort Integer. As *funções auxiliares* definem respectivamente a soma, subtração e multiplicação de dois números inteiros.

A *equação semântica* (1) define que dado um número N é retornado o seu Integer correspondente. Já, (2), (3) e (4) correspondem respectivamente a soma, subtração e multiplicação de duas expressões.

Até agora especificamos linguagens mais simples, sem declaração de variáveis ou manipulação de memória. Veremos a seguir como aperfeiçoar nosso exemplo de linguagem adicionando essas características; por meio dos modelos matemáticos proporcionados pela Semântica Denotacional.

2.1.2.2 Armazenamento - "*storage*"

O modelo abstrato usado para armazenamento de dados é chamado de *store*. Um *store* é um conjunto de *células* ou *localizações* onde cada uma delas é identificada por um endereço, ou seja, é uma representação da memória do computador. Cada posição de memória pode estar em um dos seguintes estados [Wat91]:

- *unused* - não está alocada a nenhuma variável;
- *undefined* - está alocada mas não contém nenhum valor;
- posição de memória alocada e com valor determinado.

Chamamos de *storables* todos os valores que podem ser armazenados em uma posição de memória. Existem funções auxiliares, as quais são aplicadas a um determinado *store*, para tornar possível a representação de operações tais como: alocação e desalocação de posição de memória, atualização de valores, entre outras. Vamos exemplificar como é definida a Semântica Denotacional de um comando de atribuição.

Exemplo 2.1.4 (Semântica Denotacional de um comando de atribuição) *:*

- Storable : Integer
- execute : Command → Store → Store
- evaluate : Expression → Store → Integer

(1) execute ⟦ I:Identifier ":=" E:Expression ⟧ sto =
    let val = evaluate E sto in update(sto, location I, val)

No exemplo 2.1.4 é definida a Semântica Denotacional de um comando de atribuição. Nesse exemplo uma posição de memória é atualizada de acordo com o valor resultante da expressão E, lembrando que esse valor é do tipo inteiro.

A equação semântica (1) é usada para definir a denotação dos comandos em geral, os quais atualizam a memória atual e produzem um novo armazenamento. A função semântica evaluate devolve um número inteiro a partir da expressão contida na memória atual.

A equação semântica (1) especifica o significado (denotação) do comando de atribuição. O store *sto* é alterado de acordo com o valor contido em *val*, o qual é resultado da avaliação da expressão *E*, na posição indicada pelo identificador *I*. Nessa equação semântica foram usadas duas funções auxiliares, são elas:

- location : Identifier → Location
- update: Store × Location × Storable → Store

A função location associa posição de memória a cada identificador e update retorna um novo Store, atualizado, a partir do (Store) antigo, da posição de memória específica (Location) e do novo valor (Storable).

2.1.2.3 Ambiente de avaliação - "*environment*"

O conceito de escopo é extensamente utilizado em linguagens de programação. O escopo de um identificador, de um determinado programa, é o bloco de instruções onde ele está presente e pode ser acessado [Seb00]. É importe que haja organização na definição dos nomes dos identificadores; um identificador pode representar uma variável, uma constante, assim como outros elementos do programa como: nome de procedimento, de função, tipos, etc.

Declarações estabelecem ligações entre os identificadores e entidades de um determinado sort. Essas ligações são chamadas de *bindings* [Wat91]. Ambiente, ou "*environment*" em inglês, é o conjunto de todos os bindings de um determinado bloco de instruções. O conceito de ambiente permite determinar a associação de identificadores em um determinado programa no momento da sua execução. Os tipos de valores, os quais podem ser associados aos identificadores definidos, são chamados de *bindables*.

Exemplo 2.1.5 (Ambiente com variáveis e constantes) *:*

- $env = \{\ i \mapsto 5,\ j \mapsto 0,\ k \mapsto cell0\ \}$

Acima é descrito um ambiente *env*, o qual possui os seguintes elementos: as constantes i e j, respectivamente mapeadas para os valores *5* e *0* e a variável k é o mapeamento do identificador (k) para uma posição de memória.

Na Semântica Denotacional é possível definir algumas funções auxiliares que permitem manipular esses ambientes, de forma semelhante como a especificada para o conceito de *stores*. Para que o conceito de *environment* fique mais claro, vamos exemplificar como é definida a Semântica Denotacional de uma declaração de variável.

Exemplo 2.1.6 (Semântica Denotacional de uma declaração de variável) *:*

- Bindable : Location
- Storable : Integer
- elaborate : Declaration $\rightarrow$ Environ $\rightarrow$ Store $\rightarrow$ (Environ $\times$ Store)

(1) elaborate $[\![$ "var" I:Identifier $]\!]$ env sto =
    let (sto', loc) = allocate sto in (bind(I, loc),sto')

Define-se que Bindable são as posições de memória, necessárias para que seja possível efetuar o *binding* de um determinado identificador. Os valores, os quais serão armazenados nas posições, são definidos por Storable e são do tipo inteiro.

A função semântica elaborate é usada para determinar a denotação de uma declaração de variável. Ela define que a partir de Declaration, o qual corresponde a uma declaração, Environ, representa o ambiente atual, e Store, o qual é conjunto atual das posições de memória disponíveis, é produzida uma tupla contendo um novo Environ e um novo Store, os quais representam respectivamente o mapeamento do identificador para uma posição de memória e essa posição é reservada para a variável especificada. A equação (1) expressa a semântica e o comportamento da função semântica. Ela faz uso das seguintes funções auxiliares:

- allocate : Store $\rightarrow$ Store $\times$ Location
- bind : Identifier $\times$ Bindable $\rightarrow$ Environ

A função allocate tem o papel de, dado um conjunto de posições de memória, retornar um par contendo um novo Environ e um novo Store. Por sua vez, bind retorna um novo ambiente simples, que contém apenas o Identifier e o Bindable atrelados.

Essa foi uma mera introdução aos conceitos da Semântica Denotacional, os quais serão muito úteis no desenvolvimento do trabalho, pois veremos que a semântica de ações

é uma derivação da abordagem estudada nessa seção. Para um estudo mais aprofundado é recomendado [Sch86] e para uma visão geral [Wat91].

2.1.3 Semântica Operacional

A Semântica Operacional preocupa-se em *como* executar um determinado programa e não apenas com o resultado da execução desse programa. Mais precisamente, ao usarmos a Semântica Operacional, nos interessamos em entender como os estados são modificados durante a sua execução [NN92]. A Semântica Operacional de uma determinada linguagem é definida através de *sistemas de transições* para descrever o significado dos comandos da linguagem e *definições indutivas* para expressar o conjunto de todas as transições possíveis [Win93].

A seguir veremos como a Semântica Operacional estrutural é usada, de acordo com [Win93]. Também veremos como usar a Semântica Operacional estrutural modular, segundo [Mos04].

2.1.3.1 Semântica Operacional Estrutural

Em [Plo81], Plotkin introduziu uma abordagem que descreve como um programa é executado por meio de um *sistema de transição de estados*. Tal abordagem foi chamada de **sistemas de transições**. Neste mesmo estudo é apresentado o conceito de *Semântica Operacional Estrutural* (SOS - Structural Operational Semantics), o qual enfatiza os *passos individuais* de uma determinada computação. Nesta seção iremos introduzir essa abordagem por meio de exemplos.

Abaixo é apresentada a sintaxe abstrata de uma linguagem de expressões, a qual será a base do exemplo de Semântica Operacional Estrutural a ser apresentado [Win93]:

$$a \quad ::= \quad n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1$$

Para definir a Semântica Operacional da nossa linguagem, nós precisamos definir o significado exato de um *estado*. Isto é feito através da seguinte função total:

$$\sigma : \text{Loc} \rightarrow \text{N}$$

onde é representado o mapeamento do ambiente dos identificadores e do armazenamento para os números inteiros, sendo assim é possível determinar o valor atual de uma variável.

Agora é preciso definir um método para a avaliação das expressões aritméticas, definidas na sintaxe abstrata acima, a partir de um determinado estado. Para isso nós devemos definir uma relação de avaliação entre pares e números:

$$\langle \text{a}, \sigma \rangle \rightarrow \text{n}$$

a qual tem o seguinte significado: a expressão a no estado σ é avaliada para um número inteiro n. A partir disso a avaliação de uma expressão aritmética em um dado estado é definida através de regras no estilo da *dedução natural*.

De acordo com [NN92] a Semântica Operacional *small step* possui dois tipos de configurações:

- $\langle S, s \rangle$, que representa que o comando S será computado a partir do estado s
- s, que representa um estado terminal

Usaremos a Semântica Operacional Estrutural *small step* para exemplificar a Semântica Operacional de uma linguagem de expressões aritméticas.

Exemplo 2.1.7 (SOS de uma linguagem de expressões) *:*

$$\langle n, \sigma \rangle \rightarrow n \tag{2.5}$$

$$\langle X, \sigma \rangle \rightarrow \sigma(X) \tag{2.6}$$

$$\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \qquad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 + a_1, \sigma \rangle \rightarrow n_0 + n_1} \tag{2.7}$$

$$\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \qquad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 - a_1, \sigma \rangle \rightarrow n_0 - n_1} \tag{2.8}$$

$$\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \qquad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 * a_1, \sigma \rangle \rightarrow n_0 \times n_1} \tag{2.9}$$

As regras 2.5 e 2.6 são avaliadas diretamente; qualquer número já está avaliado, sendo ele mesmo o valor, e as variáveis são avaliadas para o valor associado a elas pelo estado atual. As regras 2.7, 2.8 e 2.9 são compreendidas da seguinte maneira: se as premissas descritas acima da linha são verdadeiras, então a sua conclusão é descrita abaixo da linha. Sendo assim, no caso da regra final para $a_0 * a_1$, se no estado σ, a_0 é avaliado para n_0 e a_1 é avaliado para n_1, então nesse mesmo estado, $a_0 * a_1$ é avaliado para $n_0 \times n_1$.

2.1.3.2 Semântica Operacional Estrutural Modular

A modularidade em especificações SOS foi deixada em aberto por Plotkin em [Plo81]. Entretanto, em [Mos04] é apresentada uma versão modular da SOS, a qual foi chamada de *Semântica Operacional Estrutural Modular* (MSOS - Modular Structural Operational Semantics).

A MSOS usa um *sistema de transição generalizado*, onde componentes como ambiente e memória são implementados nos rótulos das transições em vez de serem tratados como configurações, dessa forma a modularidade é conseguida. Ou seja, na SOS, as configurações podem ser árvores sintáticas, valores computados ou entidades auxiliares, como o ambiente e a memória, por exemplo. Já na MSOS, configurações sempre serão apenas árvores sintáticas ou valores computados. Qualquer entidade auxiliar necessária deve ser implementada nos rótulos.

Na MSOS é obrigatório o uso de rótulos nas transições. Um rótulo α em uma transição representa toda a informação associada a ela, incluindo o ambiente atual e o estado da memória antes e depois da transição.

Esses rótulos das transições são vistos como morfismos de uma categoria que possuem operações de composição. Na verdade, a categoria dos rótulos é geralmente o produto de uma categoria, e uma notação é disponibilizada para acessar e alterar componentes específicos independentemente da presença de outros componentes [Mos03].

A notação usada nos rótulos é uma notação reminiscente da notação em Standard ML para *record patterns* [Mos03, dR05]. Cada parte do rótulo é definida sendo: $i = c$, isso significa que o componente c é equivalente ao índice i. A notação '...' é usada para garantir

que os componentes dos rótulos não mencionados não serão excluídos. Por exemplo, $\{\rho{=}\rho_0,...\}$ especifica um rótulo onde o componente do índice ρ é ρ_0; em $\{\sigma{=}\sigma_0,\sigma'{=}\sigma_1,...\}$ o índice σ tem como componente σ_0 e σ' tem σ_1. A meta-variável X opera sobre rótulos arbitrários e a meta-variável U é restrita a rótulos que são morfismos identidade. Pode-se escrever apenas $t \longrightarrow t'$, ao invés de escrever $t \ -U \rightarrow t'$.

Abaixo é apresentada a sintaxe abstrata de uma linguagem de expressões, similar a do Exemplo 2.1.7, com uma expressão "**let**". Essa sintaxe será usada no exemplo de uso da Semântica Operacional Estrutural Modular a ser apresentado:

$$a ::= n \mid x \mid a_0 + a_1 \mid \textbf{let}\ x = a_0\ \textbf{in}\ a_1$$

$$\text{n} \in \mathbb{N} = \{0,1,2,...\}$$
$$\text{x} \in \mathit{Var} = \{x_0,x_1,x_2,...\}$$
$$\text{a} \in \mathit{AExp}$$

Além da sintaxe abstrata, também especificamos os conjuntos necessários para a interpretação das regras. Temos que n pertence aos naturais, x ao conjunto das variáveis e a representa as expressões aritméticas. A seguir apresentamos a MSOS da sintaxe acima.

Exemplo 2.1.8 (MSOS de uma linguagem de expressões) *:*

$$\frac{U = \{\rho, ...\}, \qquad \rho(x) = n}{x - U \rightarrow n} \tag{2.10}$$

$$\frac{U = \{\rho, \sigma, ...\}, \qquad \rho(x) = l, \qquad \sigma(l) = n}{x - U \rightarrow n} \tag{2.11}$$

$$\frac{a_0 - X \rightarrow a_0'}{a_0 + a_1 - X \rightarrow a_0' + a_1} \tag{2.12}$$

$$\frac{a_1 - X \rightarrow a_1'}{n_0 + a_1 - X \rightarrow n_0 + a_1'} \tag{2.13}$$

$$\frac{n = n_0 + n_1}{n_0 + n_1 \longrightarrow n} \tag{2.14}$$

$$\frac{a_0 - X \rightarrow a_0'}{\textbf{\textit{let}}\ x = a_0\ \textbf{\textit{in}}\ a_1 - X \rightarrow \textbf{\textit{let}}\ x = a_0'\ \textbf{\textit{in}}\ a_1} \tag{2.15}$$

$$\frac{a_1 - \{\rho = \rho[n_0/x], ...\} \rightarrow a_1'}{\textbf{\textit{let}}\, x = n_0\, \textbf{\textit{in}}\, a_1' - \{\rho, ...\} \rightarrow \textbf{\textit{let}}\, x = n_0\, \textbf{\textit{in}}\, a_1'} \tag{2.16}$$

$$\textbf{\textit{let}}\, x = n_0\, \textbf{\textit{in}}\, n_1 \longrightarrow n_1 \tag{2.17}$$

As regras 2.10 e 2.11, do Exemplo 2.1.8, são referentes a avaliação de variáveis. Considerando que ρ é o ambiente e σ a memória, a avaliação de uma variável resulta em um número que pode estar armazenado diretamente no ambiente ou em uma posição de memória. Em 2.12, 2.13 e 2.14 temos as três regras necessárias para a avaliação de uma soma. Nas regras 2.15, 2.16 e 2.17 temos a avaliação de uma expressão "**let**".

Na regra 2.16 especificamente, a expressão a_1 é avaliada usando o novo ambiente (ρ) resultante da amarração da variável x ao número n. Ao usar '...' no rótulo, garantimos que os componentes não usados nessa transição também serão passados.

Uma visão geral da Semântica Operacional pode ser obtida em [Win93, NN92, Mus96]. Uma visão aprofundada da SOS é descrita em [Plo81]. Boas introduções sobre o uso da MSOS são dadas em [dOB01, dR05] e a definição completa do formalismo é encontrada em [Mos04]. Um comparativo entre as duas abordagens é dado em [Mos03].

2.2 Semântica de Ações

Especificações de linguagens de programação, por meio da sintaxe formal (usando a notação **BNF** e suas variantes) têm sido usadas e compreendidas por programadores, mas a semântica formal muitas vezes não é usada e até mesmo evitada [Wat91].

Isso acontece pelo fato de que as abordagens usadas para a definição formal de linguagens (como as apresentadas na seção 2.1) usam conceitos matemáticos, que podem ser de difícil compreensão para a maioria dos programadores e usuários de linguagens de programação, além disso, os métodos de definição da semântica de linguagens de programação costumam ter baixa escalabilidade, tornando as especificações de linguagens de uso comum de difícil compreensão, extensão e modificação [Gay02].

Por esses motivos, surgiu a idéia de criar uma técnica que melhorasse o estilo de escrita da semântica formal de linguagens de programação e ela foi denominada Semântica

de Ações. A Semântica de Ações foi originalmente desenvolvida por Peter D. Mosses [Mos92] com a colaboração de David A. Watt [Wat91]. Ela é uma derivação da semântica denotacional [Sch86, Wat91], mantendo uso de *funções semânticas* e *equações semânticas*.

A principal característica da nova abordagem é que o significado de um programa é representado por meio de uma entidade matemática chamada *ação*. Para que isso fosse possível foi criada uma notação especial para a Semântica de Ações, a qual foi chamada de *Notação de Ações* (*action notation*).

A definição de uma linguagem de programação, por meio da Semântica de Ações, é constituída pelas seguintes partes [Bro96]:

- *sintaxe abstrata* - é a especificação da gramática da linguagem;
- *semântica* - é a especificação do significado das frases da linguagem e é subdivida em duas partes:
  - *entidades semânticas* - é a especificação dos dados e operações que a linguagem manipula;
  - *funções semânticas* - é uma coleção de funções mutuamente recursivas, as quais efetuam o mapeamento dos programas para as denotações deles.

2.2.1 Sintaxe abstrata

Em [Mos92] é definida a notação padrão usada para a definição da sintaxe abstrata nas especificações por meio da Semântica de Ações.

No lado esquerdo das regras são usados símbolos não-terminais para definir os termos de cada equação da linguagem. As árvores sintáticas são definidas entre colchetes duplos ($[\![\ldots]\!]$) e por meio de aspas duplas identificamos os símbolos terminais. Como exemplo segue a sintaxe abstrata de uma linguagem de programação imperativa.

No exemplo 2.2.1 usaremos dois símbolos não-terminais: Expression define expressões aritméticas, bem como valores verdade, identificadores e números inteiros; e Command é usado para definir os comandos e declarações disponíveis nesta linguagem, como condicional, declaração de variável, laço de repetição, entre outros.

Exemplo 2.2.1 (Sintaxe abstrata de uma linguagem hipotética) *:*

grammar:

- Expression = Integer | Identifier | true | false |
  ⟦Expression "+" Expression⟧ |
  ⟦Expression "-" Expression⟧ |
  ⟦Expression "*" Expression⟧
- Command = ⟦"var" Identifier⟧ | ⟦Identifier "::=" Expression⟧ |
  ⟦"let" Identifier "in" Command "end"⟧ |
  ⟦"if" Expression "then" Command "else" Command "end"⟧ |
  ⟦"while" Expression "do" Command "end"⟧ |
  ⟦Command ";" Comnand⟧

2.2.2 Entidades semânticas

As entidades semânticas são usadas para representar o comportamento dos elementos que uma linguagem usa independente da implementação dos mesmos [Mos92]. Isso quer dizer, por exemplo, que um número inteiro é por si só um número inteiro, ou seja, a especificação de uma linguagem faz uso de números inteiros sem precisar especificar detalhadamente o seu comportamento, pois eles são entidades matemáticas bem definidas. Mais do que isso, as entidades semânticas definem os tipos de dados e as operações usadas por uma determinada linguagem, sem se preocupar com o método de implementação. Existem três tipos de entidades semânticas usadas na Semântica de Ações: ação (*action*), dados (*data*) e *yielder*.

2.2.2.1 Ação

Ação (*action*) é uma entidade computacional que pode ser executada, receber e propagar dados. Quando uma ação é executada ela pode produzir os seguintes resultados:

- complete - terminou normalmente;
- escape - terminou inesperadamente;
- fail - falhou;
- diverge - não terminou.

As ações podem ser classificadas como *primitivas* e *compostas*. As ações complete, fail e diverge são classificadas como primitivas, porque são executadas atômicamente e produzem os resultados previamente conhecidos.

As ações compostas são criadas por meio do uso de combinadores de ações (*action combinators*) e podem ser: *seqüenciadas*, *intercaladas* ou *alternativas*. Nas ações compostas seqüenciadas uma subação é executada antes da outra, nas intercaladas as subações atômicas são executadas colateralmente, ou seja, as subações atômicas são executadas em uma ordem qualquer e nas alternativas apenas uma subação é escolhida para execução, pois a escolha de uma das alternativas exclui a escolha das outras.

O fluxo de dados, na semântica de ações, é estabelecido de acordo com os tipos de dados que uma ação está manipulando (recebendo e produzindo). Esses tipos de dados são classificados como: dados transitórios (*transients*), *bindings* e posições de memória (*storage*), os quais estão detalhados a seguir.

- *transients* - são dados ou tuplas passados imediatamente entre as ações. Esses valores modelam os dados propagados pelas expressões. Eles devem ser usados imediatamente ou então serão perdidos;

- *bindings* - consiste de amarrações de identificadores (*tokens*) para dados. Eles são acessíveis durante toda a execução de uma ação e das suas subações, embora eles necessitem ser escondidos temporariamente pela criação dos escopos internos;

- *storage* - representa a memória por meio de valores armazenados em células (ou localizações). Alterações feitas no *storage* durante a execução de uma ação são persistentes, então os dados em memória podem ser alterados apenas por ações explícitas.

Exemplo 2.2.2 (Fluxo de dados na Semântica de Ações) *:*

- $\{1 \mapsto \mathsf{false}, 3 \mapsto \mathsf{cell1}, 4 \mapsto 25\}$ *são os transients;*

- $\{$“i” $\mapsto \mathsf{true}$, “j” $\mapsto \mathsf{cell2}$, “k” $\mapsto 5\}$ *são os bindings;*

- $\{\mathsf{cell1} \mapsto \mathsf{true}, \mathsf{cell2} \mapsto \mathsf{cell1}, \mathsf{cell2} \mapsto 7\}$ *é um storage.*

No exemplo 2.2.2 apresentamos como os *transients*, *bindings* e o *storage* são representados na Semântica de Ações. Os *transients* são tuplas, onde um inteiro rotula um outro *sort* de um dado qualquer; *bindings* são mapeamentos de identificadores para *bindables* (*bindables* são definidos pelo designer da linguagem) e o *storage* é um conjunto de células de memória.

De acordo com o fluxo de dados podemos classificar as ações conforme as seguintes facetas:

- *faceta básica* - trata do fluxo de controle;
- *faceta funcional* - trata das ações que processam dados transitórios (*transients*);
- *faceta declarativa* - trata das ações que geram ou recebem *bindings*;
- *faceta imperativa* - trata das ações que alteram ou consultam informações em memória (*storage*);
- *faceta reflexiva* - trata das abstrações;
- *faceta comunicativa* - trata das ações que operam sob sistemas e/ou agentes distribuídos.

Na Seção 2.2.5 apresentaremos um breve detalhamento de cada faceta citada acima, de acordo com [Mos92, Wat91, Mou93, Bro96, Mus96].

2.2.2.2 Dados

A informação processada por ações consiste em itens de dado (*data*), organizados em estruturas que controlam o seu acesso.

Dados podem ser entidades matemáticas como: valores lógicos, números, caracteres, strings, listas e mapeamentos. Eles também podem ser formados por entidades como identificadores (*tokens*), agentes e mensagens. A seguir estão descritos alguns exemplos de dados disponíveis na Semântica de Ações:

- truth-value = true | false - valores verdade podem ser true para verdadeiro e false para falso;
- integer = 0 | nonzero-integer - um número inteiro pode ser zero ou um inteiro não zero, o que compreende tanto os inteiros positivos como os inteiros negativos;
- character ≥ digit | letter - um caractere pode ser tanto um dígito, o qual corresponde a um número de 0 até 9, ou então uma letra, a qual pode ser maiúscula (a | ... | z) ou minúscula (A | ... | Z);
- bindings ≥ map[token to bindable | unknown] - é a representação do mapeamento de um identificador (token) para um valor desconhecido, o qual pode ser de qualquer sort disponível na Semântica de Ações;
- storage = map[cell to storable | uninitialized] - é a representação do mapeamento de uma célula de memória para um valor não inicializado.

2.2.2.3 Yielders

Yielders são entidades que podem produzir dados durante a execução de uma ação, tais dados produzidos são chamados de *yielded data*.

Os dados produzidos dependem das informações atuais da execução, por exemplo: os *transients* passados, os *bindings* recebidos e o atual estado da memória (*storage*). A seguir estão descritos alguns exemplos de *yielders* disponíveis na Semântica de Ações e os seus respectivos comportamentos:

- the given Y $\#n$ - produz o enésimo item de uma tupla de dados transitórios passados para uma ação, desde que seja do mesmo sort especificado por Y e n é o segundo argumento;
- the d bound to T - produz um objeto amarrado a um identificador denotado por T nos *bindings* atuais, depois verifica se o tipo é do sort especificado por d;
- the d stored in Y - produz o valor de sort d armazenado na posição de memória denotada pela célula produzida por Y.

2.2.3 Funções semânticas

As funções semânticas especificam um mapeamento entre as entidades sintáticas (árvores) e as entidades semânticas (ações). O real significado das funções semânticas são expressas por meio de equações semânticas, de maneira similar às descrições da semântica denotacional. Deve existir uma equação semântica para cada frase da linguagem.

Para melhor compreensão iremos exemplificar a função semântica dos comandos da linguagem do Exemplo 2.2.1 e a equação semântica do comando de condicional.

Exemplo 2.2.3 (Funções semânticas e equações semânticas) *:*

- execute _ :: Command → action

(1) execute ⟦"if" E:Expression "then" C_1:Command "else" C_2:Command "end"⟧ =
    | evaluate E
    then
    | | check (the given truth-value is true) and then execute C_1
    | or
    | | check (the given truth-value is false) and then execute C_2

A função semântica execute define o mapeamento de um comando para uma ação. A equação semântica (1) expressa o significado da execução de um comando condicional. Foram usadas as seguintes variáveis: E para o sort Expression; C_1 e C_2 para o sort Command. A variável E representa a expressão que será avaliada; C_1 e C_2 representam respectivamente o comando a ser executado caso a expressão seja verdadeira e o comando a ser executado caso a expressão seja falsa.

Mais precisamente, a definição do comando condicional é interpretada da seguinte maneira: primeiramente a expressão E é avaliada, logo em seguida por meio do combinador de ações then, o resultado da avaliação de E é propagado para as próximas duas subações, as quais correspondem a ação composta "A_1 or A_2"; o combinador de ações or faz com que uma das subações seja escolhida para ser executada. Finalmente é verificado o valor verdade da subação escolhida para execução, se esse valor corresponde ao propagado pela avaliação de E, então o combinador de ações and then faz com que o respectivo comando seja executado e caso contrário a outra subação é executada. Os combinadores de ações e os seus respectivos comportamentos serão apresentados na seção 2.2.5.

2.2.4 Notação de dados

A semântica de ações pré-define uma coleção de *sorts* de dados e operações relacionadas a eles. Essa coleção forma a notação de dados (*data notation*) da Semântica de Ações. Números naturais, números inteiros, caracteres, strings, listas, valores lógicos, mapeamentos e árvores são alguns exemplos de sorts da notação de dados. Além dos sorts já existentes também podemos definir nossos próprios sorts e operações por meio da álgebra unificada [Bro96].

A notação de dados padrão está descrita em [Mos92]. Vejamos alguns exemplos da notação de dados padrão:

- 1, true, "k" - são valores simples;
- truth-value, integer, list[string] - são sorts de valores e representam respectivamente: valores verdade, números inteiros e listas de string;
- product(2,3), not(true), head(["1","2","3"]) - são operações sobre dados e representam respectivamente: A multiplicação dos inteiros 2 e 3, a negação do valor verdade true e o retorno da cabeça da lista.

A especificação formal de linguagens de programação requer a definição de *entidades semânticas*. As quais são sorts particulares de informações que a linguagem manipula e operações auxiliares na especificação. As entidades semânticas costumam incluir sorts primitivos como inteiros e valores verdade, bem como sorts que definem tipos compostos, como listas e vetores. A notação de dados padrão fornece especificações de valores primitivos e compostos pré-definidos, assim como permite definir novos sorts de dados usando as definições já existentes como base.

2.2.5 Notação de Ações

A Notação de Ações é usada para escrever ações, as quais fazem parte das equações semânticas. Esta notação tem o mesmo objetivo que a notação funcional, empregada na semântica denotacional, sendo assim ela é a base para as especificações da Semântica de

Ações. A nova notação é composta por palavras da língua inglesa, as quais expressam o comportamento das ações e ainda proporcionam uma fácil leitura sem comprometer o formalismo empregado.

2.2.5.1 Faceta básica

A faceta básica preocupa-se com o fluxo de controle das ações e com a ordem em que as subações de uma ação são executadas. É na faceta básica que definimos se as ações devem ser seqüenciadas ou intercaladas. Além disso é essa faceta que especifica os tipos de terminações e confirmações que expressam o comportamento de uma ação.

A ação primitiva complete termina normalmente e com sucesso. Não produz *bindings* e não modifica o *storage*. Já a ação primitiva unfold é usada junto com o combinador unfolding para desviar o fluxo de controle ao início deste combinador. A ação "unfolding A" faz com que A substitua unfold quando ele é encontrado. O uso do combinador unfolding e da ação primitiva unfold é útil na especificação de laços de repetição.

Exemplo 2.2.4 (Faceta básica) *:*

(2) execute ⟦"while" E:Expression "do" C:Command "end"⟧ =
unfolding
| | evaluate E
then
| | execute C and then unfold
else
| | complete

Em (2) exemplificamos a execução de um comando while por meio da ação unfold e dos combinadores de ações unfolding e and then (o combinador de ações and then será introduzido a seguir). Enquanto a expressão E for verdade o comando C é executado e o laço de repetição permanece sendo executado, caso contrário o laço de repetição é terminado por meio da ação complete. O combinador de ações else é na verdade uma simplificação da seguinte ação composta:

- | | check (the given truth-value is true) and then A_1
  or
  | | check (the given truth-value is false) and then A_2

A ação funcional check verifica se o valor passado é verdadeiro ou falso. Quando a avaliação da ação "check Y" é avaliada para true (verdadeiro) a ação A_1 é executada e para o caso de false (falso) a ação A_2 é executada.

Vejamos a seguir o comportamento de algumas das principais ações compostas, as quais são construídas por meio de combinadores de ações:

- A_1 or A_2 - representa a escolha não-determinística entre as subações A_1 e A_2. A escolha de qual subação será executada é feita aleatoriamente. Todavia, se uma subação falhar a outra é executada. Se ambas as subações falharem então "A_1 or A_2" falha. Por outro lado, se uma delas completar a execução com sucesso então "A_1 or A_2" completa;

- A_1 and A_2 - faz com que a execução das subações seja intercalada, ou seja, a escolha da ordem de execução é não-determinística entre as subações atômicas de A_1 e A_2. Os dados transitórios e *bindings* de entrada são distribuídos às ações e as informações de resultado são associadas gerando um novo conjunto de dados transitórios e *bindings*, contudo o *store* final é resultante da execução da segunda subação;

- A_1 and then A_2 - permite a execução seqüencial das subações, ou seja, A_2 executa somente após o término da execução de A_1. Os dados transitórios e *bindings* de entrada são distribuídos às ações, e as informações de resultado são associadas aos dados transitórios e *bindings* anteriores, gerando um novo conjunto de dados transitórios e *bindings*.

Exemplo 2.2.5 (Faceta básica - combinadores básicos) *:*

(3) execute $[\![C_1$:Command ";" C_2:Command$]\!] =$
    | execute C_1
    and then
    | execute C_2

A equação semântica (3) exemplifica a execução seqüencial de C_1 e C_2 por meio do combinador de ações and then. O comando C_2 é executado somente após a execução completa de C_1, porque a execução de C_2 faz uso dos dados transitórios e *bindings* propagados pela execução C_1.

2.2.5.2 Faceta funcional

A faceta funcional trata das ações que processam dados transitórios (*transients*). Dados transitórios são os valores representados por um mapeamento de números, chamados rótulos (*labels*), para valores de um sort de dados (*datum*). A principal característica desta faceta é que ela recebe estes dados transitórios de ações e disponibiliza novos dados para outras ações[1].

A principal ação primitiva é "give Y", a qual avalia o *yielder* Y e produz um dado simples identificado pelo rótulo 0. Por exemplo, a partir dos seguintes dados transitórios $\{1 \mapsto 5, 2 \mapsto \mathsf{true}, 3 \mapsto \text{“goo”}\}$ temos que para:

- "give the integer #1" - o dado transitório $\{1 \mapsto 5\}$ é produzido;
- "give the truth-value #2" - o dado transitório $\{1 \mapsto \mathsf{true}\}$ é produzido;
- "give the string #5" - falhará, pois não existe a string #5 entre os dados transitórios passados a esta ação.

O combinador de ações funcional then faz com que duas ações sejam executadas seqüencialmente e os dados transitórios sejam passados entre as ações. No caso da ação composta "$A1$ then A_2", a subação A_1 é executada usando os dados transitórios de entrada, e A_2 é executada em seguida usando os dados transitórios propagados por A_1.

Exemplo 2.2.6 (Faceta funcional - combinadores funcionais) *:*

(4) evaluate ⟦E_1:Expression "*" E_2:Expression⟧ =
    | | evaluate E_1
    | and
    | | evaluate E_2
    then
    | give product(the given integer #1, the given integer #2)

A equação semântica (4) exemplifica a execução de uma multiplicação por meio do combinador funcional then. As expressões E_1 e E_2 são avaliadas e produzem os dados rotulados por 1 e 2 respectivamente. A *tupla* de dados transitórios produzidos é repassada para a subação que efetua a multiplicação dos dois valores.

[1] Existem várias versões da notação funcional de ações. Neste trabalho estamos usando a versão de [Mos92]

2.2.5.3 Faceta declarativa

Esta é a faceta que trata das ações que geram ou recebem *bindings*. Os *bindings* são representados por um mapeamento de identificadores para *bindables* (elementos do sort *bindable*). Isso caracteriza a representação do ambiente das variáveis (escopo) nas linguagens de programação. Os identificadores têm comportamento idêntico aos usados em programas, representados por *strings* de caracteres. Veremos as principais ações para manipulação de informações armazenadas em memória.

A ação indivisível "bind k to Y" avalia o *yielder* Y para um dado simples e produz um *binding*, o qual liga o identificador k ao dado resultante. No caso da ação rebind, ela apenas propaga os *bindings* atuais para a próxima ação. Por exemplo, a partir do conjunto de bindings $\{\mathsf{i} \mapsto \mathsf{true}, \mathsf{j} \mapsto 10, \mathsf{k} \mapsto \mathsf{cell1}\}$ temos que:

- "bind i to false" - produz um novo ambiente (*environment*) $\{\mathsf{i} \mapsto \mathsf{false}\}$;
- "rebind" - propaga os bindings atuais $\{\mathsf{i} \mapsto \mathsf{true}, \mathsf{j} \mapsto 10, \mathsf{k} \mapsto \mathsf{cell1}\}$;
- "give the integer bound to "j"" - produz o dado transitório $\{1 \mapsto 10\}$.

Vejamos a seguir o comportamento de alguns dos principais combinadores declarativos:

- furthermore A - executa a ação A e produz os *bindings* recebidos, atualizados com os bindings da ação A;
- A_1 hence A_2 - executa seqüencialmente as subações A_1 e A_2, de tal forma que os *bindings* de A_1 são propagados para A_2 para produzir a informação final;
- A_1 moreover A_2 - executa ambas as subações A_1 e A_2 de maneira intercalada. Quando a execução é completa, os *bindings* produzidos pelas subações são unidos, mas prevalecem os gerados pela segunda ação.
- A_1 before A_2 - executa as subações A_1 e A_2 seqüencialmente, de tal forma que A_2 é executada apenas quando A_1 termina a execução normalmente. A subação A_1 é executada com os *bindings* originais do ambiente e A_2 é executada com os *bindings*

originais mais os produzidos por A_1. Ao final da execução, os *bindings* produzidos pelas subações são unidos, mas prevalecem os gerados pela segunda ação. No geral é o mesmo comportamento que "A_1 and A_2".

Exemplo 2.2.7 (Faceta declarativa - combinadores declarativos) *:*

(5) execute ⟦"let" I:Identifier "in" C:Command "end"⟧ =
    | furthermore evaluate I
    hence
    | execute C

Na equação semântica (5) primeiro avaliamos o identificador I. Depois os *bindings* recebidos pelo bloco de comando, sobrecarregados pelos *bindings* produzidos pela avaliação de I, são produzidos para a execução de C.

2.2.5.4 Faceta imperativa

A faceta imperativa é responsável pelas ações que alteram ou consultam informações em memória (*storage*). Nesta faceta existem operações para manipulação de memória como: alocação e liberação de células de memória e armazenamento de informações.

As operações efetuadas no *storage* têm efeito apenas na célula escolhida, isto se deve ao fato de as células de memória serem independentes umas das outras. A representação da memória, segunda a Semântica de Ações, é um conjunto potencialmente infinito de células, as quais são denominadas *cells* e os valores nelas armazenados são formalmente definidos como *storbles*.

Uma posição de memória é representada por um mapeamento de uma célula (*cell*) para um valor armazenado (*storable*). Por exemplo, a partir da seguinte representação de *storage* $\{\mathsf{cell1} \mapsto \mathsf{false}, \mathsf{cell2} \mapsto \mathsf{cell1}, \mathsf{cell3} \mapsto 1\}$ temos que:

- "store true in cell1" - produz o novo *store* $\{\mathsf{cell1} \mapsto \mathsf{true}, \mathsf{cell2} \mapsto \mathsf{cell1}, \mathsf{cell3} \mapsto 1\}$, agora com o valor (*storable*) true na célula cell1;
- "deallocate cell2" - desaloca a célula cell2;
- "give the integer stored in cell3" - retorna o valor 1.

A ação primitiva "allocate Y" aloca arbitrariamente uma célula no *store* atual com o estado *undefined* [Wat91], indicando que a célula foi alocada mas não contém nenhum valor armazenado. Quando uma célula não foi alocada e não contém nenhum valor armazenado dizemos que o seu estado é *unused*. Na verdade, a ação "allocate a cell" é uma abreviação da seguinte ação híbrida [Mos92]:

- indivisibly
  | choose a cell [not in the mapped-set of the current storage] then
  | | reserve the given cell and give it

Uma ação híbrida é uma ação que usa mais de um tipo de faceta. No caso de allocate, ela faz uso das facetas imperativa e funcional.

Exemplo 2.2.8 (Faceta imperativa + faceta funcional) *:*

(6) execute ⟦"var" I:Identifier⟧ =
  | allocate a cell
  then
  | bind I to cell #1

A equação semântica (6) exemplifica a declaração de uma variável. Uma célula de memória é alocada com rótulo 1. Um novo ambiente é gerado e contém um *binding* do identificador I, o qual é amarrado a uma posição de memória.

Para liberarmos uma posição de memória alocada usamos a ação "deallocate C". Essa ação desaloca a posição de memória representada por C, todavia caso a posição de memória C não esteja alocada, então a ação falha.

Após alocarmos uma posição de memória usamos a ação "store Y_1 in Y_2" para armazenar um determinado valor na célula alocada. O *yielder* Y_1 é avaliado e produz um dado simples d, já o *yielder* Y_2 é avaliado e produz uma célula de memória c. A ação então atualiza o conteúdo da célula c com o valor d. Se qualquer uma das subações falhar, então a ação store falha. Da mesma forma, se a célula c não pertence ao domínio do *storage* atual, ou se o dado d não é do sort *storable*, então a ação store falha [Bro96].

Exemplo 2.2.9 (Faceta declarativa - combinadores imperativos) *:*

(7) execute ⟦I:Identifier "::=" E:Expression⟧ =
  | evaluate E
  then
  | store the given integer in the cell bound to I

A equação semântica (7) especifica a semântica de um comando de atribuição. Ela avalia a expressão E e retorna um inteiro, o qual é propagado para a segunda subação onde é armazenado na posição de memória referenciada pelo identificador I.

2.2.5.5 Faceta reflexiva

A faceta reflexiva é responsável pelas abstrações. Uma abstração é um elemento de dado, o qual encapsula uma ação. Abstrações podem ser propagadas como dados transitórios, amarradas a identificadores e armazenadas em posições de memória, assim como qualquer outro tipo de dado.

Funções e procedimentos de linguagens de programação são representados por meio do uso de abstrações. A ação encapsulada na abstração é normalmente executada em um contexto diferente daquele em que ela mesma ocorre [Mos92].

A ação "enact Y" avalia o *yielder* Y e produz uma abstração A. Ela então executa a ação encapsulada A com os dados transitórios e *bindings* fornecidos pela operação closure. Se a execução da ação encapsulada falhar, então a ação enact também falha.

A operação "closure Y" avalia o *yielder* Y e produz uma abstração A. O seu resultado é um dado, definido a partir do encapsulamento da ação, e a informação sobre o *binding* recebido.

2.2.5.6 Faceta comunicativa

Agentes e sistemas distribuídos são tratados pela faceta comunicativa. Esta faceta é usada para expressar o comportamento da execução concorrente por meio de um sistema distribuído de agentes, onde cada agente pode comunicar-se com outros agentes enviando e recebendo mensagens. Um agente representa a identidade de um simples processo, embutido em uma comunicação universal [Mos92].

A comunicação entre agentes é *assíncrona*. Um agente envia uma mensagem para outro agente, especificado como receptor, mas não espera ela ser entregue, pois supõe-se que uma mensagem nunca é perdida.

A ação primitiva "send Y", onde o *yielder* Y produz um sort mensagem (*message*), efetua a transmissão de uma mensagem.

Nesta seção apresentamos uma breve introdução ao formalismo da Semântica de Ações. Sua definição completa é encontrada em [Mos92] e uma visão geral é apresentada em [Wat91]. Boas introduções bibliográficas, à Semântica de Ações, são expostas nos seguintes trabalhos: [Mou93, Bro96, Mus96, Car02, Ara04].

2.3 Modularidade em Semântica de Ações

A Semântica de Ações é um formalismo usado para descrever a semântica de linguagens de programação. No entanto possui estruturas muito inflexíveis, o que acaba tornando a descrição de uma linguagem real em uma tarefa difícil, pois a definição de uma linguagem é feita em um único documento e dividida em dois grandes blocos: *sintaxe abstrata* e *semântica*.

Um problema desse tipo de formalismo é que a definição de um conceito é dividida em duas partes (sintaxe e semântica) e em pontos diferentes da especificação tornando a manipulação deles extremamente difícil, pois na definição de uma linguagem de programação real esses pontos podem estar separados por diversas páginas. O fato de ser difícil de identificar segmentos da especificação (ex: funções) é outro problema.

Para solucionar esses problemas e possibilitar o reúso de descrições já existentes foram propostas novas abordagens: *Semântica de Ações baseada em módulos*, *Semântica de Ações baseada em componentes* e *Semântica de Ações Orientada a Objetos*. As quais são apresentadas a seguir.

2.3.1 Semântica de Ações baseada em módulos

Definições de linguagens de programação devem ser modularizadas, pois dessa forma o trabalho dos desenvolvedores de linguagens concentra-se em definir e reutilizar vários módulos alternativos, possibilitando a escolha do melhor método e a rejeição dos inconsistentes [DM01].

A Semântica de Ações baseada em módulos é um método para compor linguagens pelo reúso, extensão e combinação de módulos em Semântica de Ações e é apresentado por Doh e Mosses em [DM01]. Cada módulo é especificado separadamente, e então um novo módulo de uma linguagem é definido pela combinação de outros módulos existentes.

2.3.1.1 Notação de módulos

A notação de módulos, usada em [DM01], foi definida para as expressar as especificações da Semântica de Ações baseada em módulos. A especificação de um módulo é composta por duas partes: sintaxe abstrata e semântica. A sintaxe abstrata define a estrutura da linguagem e a semântica é encarregada de expressar o significado da mesma. A seguir mostraremos alguns exemplos, adaptados de [DM01], de uso deste formalismo para a criação de módulos de uma linguagem de expressões aritméticas:

```
module Expression
    syntax:
        Exp
    semantics:
        datum >= expressible
        evaluate ⟦ Exp ⟧ → Action %% giving an expressible
```

O módulo Expression define o *sort* sintático Exp. Neste ponto da especificação é explicitada apenas a existência do sort, mas não os seus valores. Na semântica desse módulo temos o sort datum, o qual é primitivo da notação, representando todos os itens de dados que não são tuplas. Por meio do operador ">=" introduzimos o sort expressible, o qual usaremos nas expressões, sendo um subsort de datum. A função semântica evaluate define o mapeamento da entidade semântica Exp para uma ação, a qual produz um expressible, como é mostrado no comentário (iniciado com %%) da última linha da definição. A seguir está exemplificado o módulo que apenas define a assinatura da operação:

```
module ArithmeticOperator
    syntax:
        AOp
    semantics:
        datum >= expressible %% taking (expressible, expressible)
        operate ⟦ AOp ⟧ → Action %% giving an expressible
```

O módulo ArithmeticOperator define o sort sintático AOp, o qual será usado para definir as operações aritméticas. A função semântica operate define o mapeamento da entidade semântica AOp para uma ação, a qual recebe dois valores e produz o resultado da operação efetuada.

Tomando como base os módulos Expression e ArithmeticOperator, podemos usá-los para definir novos módulos para nossa linguagem de expressões aritméticas:

```
module ArithmeticExpression
    imports Expression ArithmeticOperator
    syntax:
        Exp ::= Num | ⟦ Exp AOp Exp ⟧
        Num ::= Int
        E1, E2 → Exp
        N → Num
    semantics:
        expressible >= integer
        [1] evaluate N = give the integer N
        [2] evaluate ⟦ E1 AOp E2 ⟧ = (evaluate E1 and evaluate E2) then
                                      operate ⟦ AOp ⟧
```

No módulo acima, incorporamos as definições de Expression e ArithmeticOperator ao usarmos a palavra chave imports antes do nome dos módulos em questão. O novo módulo ArithmeticOperator é definido e a sua sintaxe determina que uma expressão pode ser tanto um número inteiro (Num) ou uma operação entre expressões (Exp AOp Exp). Também são definidas as variáveis E1 e E2 para representar os números de uma possível operação; N representa um número inteiro já avaliado. Os colchetes duplos (⟦ ... ⟧) na sintaxe identificam uma árvore sintática.

A semântica desse módulo também faz uso do operador ">=", agora ele define que integer é um subsort de expressible, o último foi definido em Expressions. Além disso o módulo possui duas equações semânticas, uma para cada caso da definição sintática que aparece no mesmo módulo.

Em [1] mapeamos a entidade sintática N para um número inteiro; já em [2] definimos que os resultados da avaliação das duas subexpressões (E1 e E2) são propagadas para a função operate, a qual é responsável por efetuar uma determinada operação entre dois inteiros, os exemplos a seguir definem as operações disponíveis:

```
module ArithmeticOperatorSum
    imports ArithmeticOperator
    syntax:
        AOp ::= "+"
    semantics:
        expressible >= integer
        [3] operate ⟦ + ⟧ = give (the int #1 + the int #2)

module ArithmeticOperatorDifference
    imports ArithmeticOperator
    syntax:
        AOp ::= "-"
    semantics:
        expressible >= integer
        [4] operate ⟦ - ⟧ = give (the int #1 - the int #2)

module ArithmeticOperatorProduct
    imports ArithmeticOperator
    syntax:
        AOp ::= "*"
    semantics:
        expressible >= integer
        [5] operate ⟦ * ⟧ = give (the int #1 * the int #2)
```

O módulo ArithmeticOperationSum representa a soma entre dois valores, a subtração é expressa em ArithmeticOperationDifference e ArithmeticOperationProduct é o módulo responsável pela descrição da multiplicação. Os três módulos possuem a mesma estrutura e a principal diferença entre eles é o operador usado para executar a operação aritmética desejada.

A sintaxe especifica qual é o operador de cada módulo. Na semântica temos a implementação da equação semântica de cada operação. A soma é definida em [3], a subtração em [4] e a multiplicação em [5]. Todas as operações recebem dois inteiros e produzem um novo inteiro como resultado.

Em [DM01] é usado o formalismo ASF+SDF para expressar descrições por meio da Semântica de Ações. O ASF+SDF nasceu do casamento de dois outros formalismos: ASF (*Algebric Specification Formalism*) e SDF (*Syntax Definition Formalism*). ASF é baseado na definição de um módulo que declara uma assinatura (sorts, símbolos de funções e variáveis) e proporciona um conjunto de equações condicionais as quais definem as suas propriedades. Módulos podem ser importados em outros módulos. SDF permite a

definição da sintaxe concreta (léxica e livre de contexto) para operações e variáveis. Mais detalhes sobre esse formalismo são obtidos em [vdBHdJ+01, DHK96].

2.3.1.2 Combinando módulos

Após definirmos vários módulos podemos criar uma linguagem apenas combinando esses módulos já especificados. Também podemos usar módulos definidos em outros projetos para iniciar um novo projeto, apenas importando os módulos desejados.

Como exemplo iremos definir a linguagem ArithmeticLanguage, a qual combina os módulos exemplificados na seção anterior. A especificação da linguagem é feita pelo módulo (composto) exemplificado a seguir:

```
module ArithmeticLanguage
    imports
        ArithmeticExpression
        ArithmeticOperatorSum
        ArithmeticOperatorDifference
        ArithmeticOperatorProduct
```

O módulo ArithmeticLanguage importa o módulo ArithmeticExpression, o qual inclui Expression e ArithmeticOperator. Os módulos das operações aritméticas também são inclusos.

Da mesma maneira que definimos ArithmeticLanguage, podemos definir por exemplo: BooleanLanguage ou RelationalLanguage (não exemplificados neste documento) para especificar expressões booleanas e relacionais, respectivamente. Dessa forma é possível, por exemplo, criar uma linguagem de expressões apenas combinando módulos.

Entretanto, há conflito entre módulos quando existem duas equações semânticas para o mesmo construtor sintático, ou seja, quando incluimos sorts de determinada maneira que S1 → S2 e S2 → S1 aparecem juntos em um módulo combinado, isso implica que os sorts são idênticos, incluindo exatamente os mesmos valores e as seguintes inconsistências são geradas: duas ações podem falhar sem motivo ou então os módulos não são unificáveis. Módulos são combinados satisfatóriamente desde que não haja conflito entre as equações semânticas.

Quando existem duas equações semânticas para o mesmo construtor sintático nos módulos que estão sendo combinados, dizemos que existe um *conflito*. Tais conflitos podem ser resolvidos automaticamente ao unificar essas ações, com equações semânticas conflitantes, usando "(tentatively A_1) otherwise A_2" para determinar uma escolha entre elas.

A execução da combinação de ações, descrita acima, primeiramente executa A_1 e não executa A_2 se A_1 terminou normalmente, mas executa A_2 (com os mesmos dados de A_1) no caso de A_1 falhar.

2.3.2 Semântica de Ações baseada em componentes

A Semântica de Ações baseada em componentes define um novo estilo para descrições de linguagens por meio da Semântica de Ações. Esta nova abordagem pode ser usada para construir descrições, de linguagens de programação, mais flexíveis e reutilizáveis [MM01].

O formalismo Semântica de Ações baseada em módulos foi proposto por Menezes e Moura em [MM01] e é na verdade uma extensão da Semântica de Ações [Mos92]. A nova abordagem define duas novas notações para descrever linguagens de programação, são elas: a *notação de componentes* e a *notação de linguagens de programação*.

2.3.2.1 Notação de componentes

A especificação de um componente, da mesma maneira que a de um módulo de [DM01], é composta por duas partes: sintaxe e semântica. A notação de componentes é o agrupamento da sintaxe e da semântica e permite que estruturas reutilizáveis sejam definidas.

A seguir mostraremos alguns exemplos, originários de [MM01], para apresentar a notação de componentes. O primeiro exemplo é um componente genérico, denominado binary, para operações binárias sobre expressões aritméticas.

O componente binary recebe três parâmetros: o tipo da expressão, a string que corresponde ao operador e o *yielder* que indica como combinar os resultados produzidos pelas subexpressões:

- binary _ _ _ :: component, string, yielder → component

```
binary c s y =
    syntax
        ⟦ c s c ⟧
    semantics
        | | semantics of the c defined by the subtree #1
        | and
        | | semantics of the c defined by the subtree #2
        then
        | give y
```

A sintaxe é especificada entre colchetes duplos (⟦ ... ⟧) e determina que uma operação binária é formada por um componente c, seguida de um string s para determinar o operador e novamente outro componente c; os componentes estão representando as expressões que serão executadas pelo operador contido em s.

Na parte da semântica expressamos como uma operação binária ocorre, ou seja, dois componentes contendo as respectivas expressões são avaliados colateralmente, por meio do combinador de ações and, após isso os respectivos valores das expressões são propagados por then para a ação give, a qual é responsável por efetuar a operação contida em s. Para que isso fosse possível, usamos três novos operadores definidos pela notação de componentes:

- semantics of c - determina a ação de um determinado componente c;
- c defined by s - a partir de um componente c e de uma árvore sintática s produz um novo componente;
- subtree $\#n$ - retorna a enésima subtree do componente atual.

O componente binary serve de base para a definição de outros componentes e pode ser usado para definir uma linguagem de expressões em uma notação mais compacta, como a definida no exemplo que segue:

```
Expression = ... ++
    binary Expression "+" (the sum of them) ++
    binary Expression "-" (the difference of them) ++
    binary Expression "*" (the product of them) ++
    ...
```

Neste exemplo, os operadores da linguagem são definidos usando o construtor do componente binary. O operador "++" é usado para compor componentes. O uso deste operador, no exemplo acima, permite a definição do componente Expression, resultado da composição dos componentes de cada operador disponível na linguagem. A maior vantagem desse estilo, Semântica de Ações baseada em componentes, é que a definição da sintaxe e da semântica estão no mesmo ponto da especificação.

2.3.2.2 Notação de linguagens de programação

A notação de linguagens de programação suporta a criação, uso e extensão da especificação de uma linguagem de programação, pois apenas a notação de componentes não nos permite representar uma linguagem por completo [MM01]. A notação desta seção permite criar descrições estendíveis de linguagens de programação e define ações de descrições que são usadas e compostas para definir uma linguagem.

O operador primitivo desta abordagem é "extends [n] with c", o qual estende a definição do componente identificado por [n] com o componente c. Para estender as definições de componentes o operador pode executar as seguintes tarefas:

- se não existe nenhuma definição para o componente, n é definido da mesma maneira que o componente c;
- caso contrário, a definição existente será combinada com c usando o operador ++.

Para exemplificar o uso do operador primitivo, o último exemplo da seção anterior pode ser convertido diretamente para a notação de linguagens de programação, resultando a seguinte especificação:

```
Expression =
    extends [ Expression ] with
        binary Expression "+" (the sum of them) ++
        binary Expression "-" (thedifference of them) ++
        binary Expression "*" (the product of them) ++
        ...
```

O exemplo acima define a linguagem Expression por meio do operador extends, o qual estende um componente com nova especificação. Quando o componente não existe, este

operador tem por função criá-lo, como é o caso do exemplo citado quando ele descreve as três diferentes maneiras de utilizar o componente binary para soma, subtração ou multiplicação de expressões e eles são compostos pelo operador "++". O resultado final de extends é uma linguagem.

Descrições de linguagens mais complexas podem ser criadas a partir do uso do operador "l_1 and l_2". A linguagem resultante é a combinação das linguagens l_1 e l_2, como a descrita no exemplo a seguir:

```
ImpExp =
    | Expression
    and
    | extends [ Identifier ] with
    | | syntax ⟦ Identifier ⟧
    | | semantics
    | | | give the value stored in the cell bound to the subtree #1
    | | or
    | | | give the value bound to the subtree #1
```

A linguagem ImpExp é criada pela composição de Expression e Identifier. O primeiro componente é responsável pelas operações aritméticas e o segundo trata dos valores armazenados em posições de memória, os quais são representados por identificadores.

2.3.3 Semântica de Ações Orientada a Objetos

A Semântica de Ações Orientada a Objetos (SAOO) é uma abordagem que visa a divisão de especificações em módulos ou componentes em Semântica de Ações, adicionando algumas facilidades advindas da área de orientação a objetos [Car02].

As abordagens de módulos e componentes foram apresentadas anteriormente. A primeira delas (Semântica de Ações baseada em módulos) têm a vantagem de usar a versão da Notação de Ações, já definida, sem introduzir novos construtores para ações; uma nova sintaxe é definida apenas para definir módulos. Entretanto, a abordagem de módulos pode apresentar problemas em alguns casos, já que a localização das definições não é tratada pelo formalismo. Por outro lado, a segunda abordagem (Semântica de Ações baseada em componentes) não tem o problema citado, mas muda significativamente a sintaxe das ações, bem como o uso delas.

A Semântica de Ações Orientada a Objetos foi proposta por Carvilhe em [Car02, CM03] com o intuito de resolver os problemas das abordagens anteriores e consiste em uma *hierarquia* de classes definida a partir da estrutura sintática da linguagem. O objeto representa cada frase (declaração, comando, expressão) da linguagem, sendo assim pode existir mais de uma representação de objeto para a mesma frase.

Assim como na abordagem de módulos [DM01] e na de componentes [MM01], na orientada a objetos uma classe é composta por duas partes básicas: sintaxe e semântica. O uso de objetos requer algumas modificações no estilo da escrita de especificações. Vejamos a seguir um exemplo de uso do conceito apresentado:

Class Expression
    syntax:
        Exp
    semantics:
        evaluate ⟦ _ ⟧ : Exp → Action
End Class

No exemplo acima definimos a classe Expression para representar as expressões aritméticas de uma linguagem e pode ser usada como base para as próximas definições, como por exemplo: soma, subtração e multiplicação.

Na sintaxe, da classe, definimos o sort sintático Exp. Na semântica temos a função semântica evaluate, a qual define o mapeamento da entidade semântica Exp para uma ação. Na Semântica de Ações Orientada a Objetos consideramos as funções e equações semânticas como os métodos da orientação a objetos. Por isso, evaluate é um método de Expression e pode ser sobrecarregado pelas classes derivadas de Expression.

A partir da classe definida anteriormente, podemos usá-la para definir novas classes derivadas de Expression, para nossa linguagem de expressões aritméticas:

Class ExpressionSum
    extending Expression
    using E_1:Expression, E_2:Expression
    syntax:
        Exp ::= E_1 "+" E_2
    semantics:
        evaluate ⟦ E_1 "+" E_2 ⟧ =
            evaluate E_1 and evaluate E_2
            then give sum(the integer #1, the integer #2)
End Class

A diretiva extending indica que a classe ExpressionSum é uma subclasse da superclasse Expression. Os objetos E_1:Expression e E_2:Expression são instânciados pelo uso da diretiva using. Esses objetos são responsáveis por expressar as duas expressões que serão somadas.

O sort sintático Exp é redefinido na sintaxe da nova classe para a soma de duas expressões. Na semântica o método evaluate é sobrecarregado e a Semântica de Ações da soma de duas expressões aritméticas é especificada para ele. Se definíssemos as respectivas classes para subtração e multiplicação, então teríamamos a seguinte hierarquia de classes:

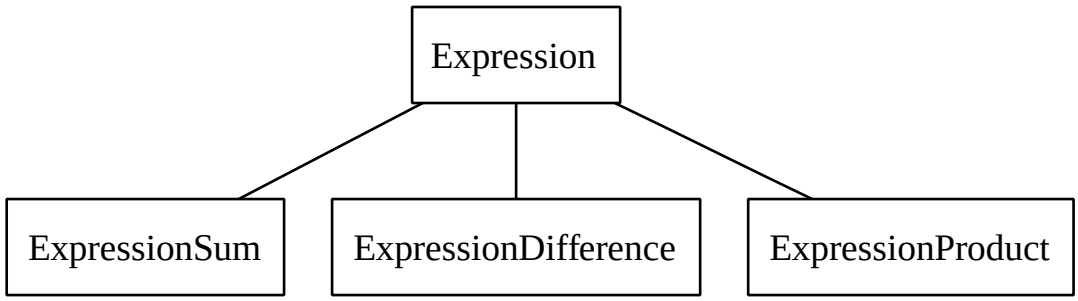

Figura 2.1: Hierarquia de classes

Na Semântica de Ações Orientada a Objetos, Carvilhe [Car02] continua usando a Notação de Ações de Mosses [Mos92]. Porém, vimos que há uma nova interpretação para as funções semânticas, ou seja, na Semântica de Ações a ação "evaluate E" é expressa por equações semânticas sobre a sintaxe contida em E. Enquanto na nova abordagem, evaluate é tratado como um método que opera sobre o objeto E, possuindo a mesma estrutura da equação semântica e produzindo a mesma ação. Para expressar esse conceito, Carvilhe define em [Car02] a sintaxe e a semântica do novo formalismo.

2.3.3.1 Sintaxe

Uma especificação por meio da Semântica de Ações Orientada a Objetos é composta por um conjunto finito de classes. Diretivas que indicam os objetos instanciados são responsáveis por formar o relacionamento entre as classes, bem como a hierarquia delas. A estrutura das frases de uma linguagem é dada pela especificação de sua árvore sintática. Pela Notação de Ações [Mos92] é expressa a semântica. Apresentaremos a seguir, de forma

resumida, a notação usada na Semântica de Ações Orientada a Objetos:

(1) Class-Module =
⟦ "Class" Class-Name Class-Body "End-Class" ⟧

(2) Class-Body =
⟦ <"extending" Base-Class-Name>?
<"using" Objects-Declaration>?
<Class-Definition>? ⟧

(3) Base-Class-Name =
⟦ Class-Name | Class-Name "::" Base-Class-Name ⟧

(4) Objects-Declaration =
⟦ Object-Declaration ⟧ | ⟦ Object-Declaration "," Objects-Declaration ⟧

(5) Object-Declaration =
⟦ Identifier ":" Class-Name ⟧

Definimos a declaração de uma classe pelo token Class (1). Em seguida especificamos o nome da classe, o seu corpo e a definição da classe é encerrada pelo token End-Class. Podemos indicar a classe-base e os objetos por ela utilizados no corpo da classe (2), a partir do uso das seguintes diretivas:

- extending - permite que a classe atual herde os atributos de uma outra classe (classe-base) (3), dessa forma uma hierarquia de classes é criada. O acesso a uma determinada classe dentro da hierarquia é representado pelo operador de escopo "::";
- using - Instancia objetos (4) para serem usados na classe atual. Cada objeto instanciado é representado por um identificador seguido de ":" e da classe correspondente (5).

A seguir apresentamos a notação para a definição de classes:

(6) Class-Definition =
⟦ "syntax" ":" Syntactic-Part "semantics" ":" Semantic-Part ⟧

(7) Syntactic-Part =
⟦ TokenName | TokenName "::=" syntax-tree ⟧

(8) Semantic-Part =
⟦ < Semantic-Functions >? < Semantic-Equations >? ⟧.

(9) Semantic-Functions =
⟦ Semantic-Function ⟧ | ⟦ Semantic-Function Semantic-Functions ⟧

(10) Semantic-Function =
⟦ Function-Name "␣"* Tokens "->" "Action" | data ⟧

(11) Tokens =
⟦ TokenName ⟧ | ⟦ TokenName "," Tokens ⟧

(12) Semantic-Equations =
⟦ Semantic-Equation ⟧ | ⟦ Semantic-Equation Semantic-Equations ⟧

(13) Semantic-Equation =
⟦ Function-Name "[[" syntax-tree "]]" "=" Action ⟧

A especificação de uma classe (6) é formada pelas seções de sintaxe e semântica, descritas a seguir:

- syntax - É aqui que a sintaxe abstrata da frase corrente é especificada. A parte sintática (7) pode ser apenas um identificador ou o mesmo seguido de uma árvore sintática (syntax-tree) como em [Mos92];

- semantics - A semântica (8) das frases é expressa por meio de funções semânticas e equações semânticas, as quais são tratadas como métodos neste novo formalismo. Usamos as regras (9), (10) e (11) para determinar as funções semânticas. Sendo que uma função semântica é um mapeamento de entidades sintáticas (Token) para uma ação ou dado (Acion ou data). As equações semânticas são definidas em (12) e (13), onde um nome de função (Function-Name) é associado a uma determinada árvore sintática (syntax-tree) e a sua denotação é uma ação.

E finalmente é apresentada a sintaxe da notação usada para definir a semântica das frases (corpo dos métodos):

(14) Action =
⟦ "complete" ⟧ | ⟦ "fail" ⟧ | ⟦ "unfold" ⟧ |
⟦ Action "and" Action ⟧ | ⟦ Action "and then" Action ⟧ |
⟦ "unfolding" Action ⟧ | ⟦ "give" Yielder "label" "#" n ⟧ |
⟦ "check" Yielder ⟧ | ⟦ Action "then" Action ⟧ |
⟦ "bind" Yielder "to" Yielder ⟧ |
⟦ "furthermore" Action ⟧ | ⟦ Action "hence" Action ⟧ |
⟦ Action "moreover" Action ⟧ | ⟦ Action "before" Action ⟧ |
⟦ "store" Yielder "in" Yielder ⟧ | ⟦ "deallocate" Yielder ⟧ |
⟦ "enact" Yielder ⟧ | ⟦ Action "else" Action ⟧ |
⟦ "recursively" "bind" Yielder "to" Yielder ⟧ |
⟦ "allocate a" Sort ⟧

(15) Yielder =
⟦ "the" Sort "#" n ⟧ |
⟦ "the" Sort "bound" "to" k ⟧ |
⟦ "the" Sort "stored" "in" Yielder ⟧ |
⟦ Yielder "with" Yielder ⟧ |
⟦ "closure" Yielder ⟧ |
⟦ "abstraction of" Action ⟧

(16) Sort =
⟦ "bindable" ⟧ | ⟦ "cell" ⟧ | ⟦ "cell" "[" Sort "]" ⟧ |
⟦ "storable" ⟧ | ⟦ "abstraction" ⟧ | ⟦ "datum" ⟧ |
⟦ "integer" ⟧ | ⟦ "value" ⟧ | ⟦ "truth-value" ⟧ |
⟦ Sort " | " Sort ⟧

Com base em [Mou93], Carvilhe introduz as regras (14), (15) e (16) para definir a sintaxe da Notação de Ações e dados. Em (14) são definidas as ações, os *yielders* são definidos em (15) e em (16) são apresentados os *sorts* usados pela Semântica de Ações Orientada a Objetos.

2.3.3.2 Semântica

A hierarquia entre classes é possível mediante a aplicação dos conceitos de orientação a objetos. Uma superclasse sempre é mais genérica que uma subclasse, pois a superclasse concentra os conceitos mais abstratos enquanto a subclasse implementa novos conceitos derivados dos herdados. Mediante isso foi criada a classe mãe de todas, a qual é uma classe-base e é a mais genérica de todas, chamada *State*. É esta super-classe a base para a criação de qualquer outra classe.

A super-classe *State* possui atributos genéricos correspondentes a dados transitórios (*transients*), *bindings* e armazenamento (*storage*) e também disponibiliza operações para interagir com esses atributos. As subclasses herdam os atributos e operações definidas na super-classe. Veremos a seguir uma breve apresentação da classe principal das especificações em Semântica de Ações Orientada a Objetos.

A classe *State*

Na Semântica de Ações Orientada a Objetos, a semântica de uma linguagem é representada por uma estrutura de classes. As classes definidas neste formalismo são subclasses

diretas de *State*, a qual é responsável por representar a Semântica de Ações original [Mos92]. Ao usar os conceitos de objetos permitimos o reaproveitamento e a extensão de especificações.

Atributos da classe *State*

Os atributos são responsáveis pelos tipos de informações usadas nas especificações de linguagens. A classe *State* possui três atributos: dados transitórios (*transients*), *bindings* e armazenamento (*storage*).

Operações sobre a classe *State*

Na Semântica de Ações Orientada a Objetos são disponibilizadas algumas operações para a modificação dos atributos de *State*. Essas operações são classificadas de acordo com as facetas da Semântica de Ações, definida em [Mos92], e usam os mesmos termos dela. Vejamos a seguir a classificação de ações que as operações seguem:

- *básicas* - controle de fluxo;
- *funcionais* - *transients*;
- *imperativas* - *storage*;
- *declarativas* - *bindings*;
- *reflexivas* - abstrações;
- *híbridas* - comportamento múltiplo.

Não são definidas operações para a faceta comunicativa. A semântica formal de cada operação, bem como os seus exemplos de uso com a Semântica de Ações, estão em [Car02].

2.4 A LFL

Vimos nas seções anteriores alguns métodos para a especificação formal de linguagens. Os primeiros formalismos eram extremamente matemáticos e a criação, compreensão e

modificação de especificações, que usavam esses formalismos, eram difíceis. Por isso, novas técnicas foram desenvolvidas com o passar do tempo e sempre visando uma leitura e escrita mais fácil.

Da semântica denotacional nasceu a Semântica de Ações, a qual serviu de base para a criação da Semântica de Ações Orientada a Objetos. A última usa conceitos de orientação a objetos em especificações de Semântica de Ações, permitindo a reutilização e extensão de código.

No desenvolvimento da especificação de uma determinada linguagem de programação é comum encontrarmos partes da especificação que possuem conceitos semelhantes. Quando isso acontece é interessante reutilizar estas partes, ao invés de ter código duplicado. Para que a reutilização de código seja possível deve-se adotar uma estrutura comum que agrupe essas informações compartilhadas. O conceito de classe pode ser adotado para tal estruturação.

Em [Ara04, AM04], Araújo propõe um modelo para a criação de uma biblioteca de classes. A qual foi denominada LFL (*Language Features Library*). Sua função é reunir e organizar classes genéricas (especificação similar) em uma estrutura, funcionando como um repositório de classes e disponibilizando formas de acesso a essas classes [Ara04].

A seguir apresentaremos uma breve introdução a estrutura organizacional da LFL, bem como a sua sintaxe.

2.4.1 Estrutura Organizacional

Uma estrutura de árvore foi adotada para representar a organização das classes na LFL, onde os níveis de hierarquia são representados por nodos e as classes por folhas. A LFL é um conjunto de classes da Semântica de Ações Orientada a Objetos e está, hierarquicamente, ligada à classe *State*.

Como representado na Figura 2.2, a LFL é composta por três estruturas respectivamente ligadas às principais estruturas de um especificação em Semântica de Ações: sintaxe (*Syntax*), semântica (*Semantics*) e entidades semânticas (*Entity*). A classe principal da biblioteca é uma subclasse de *State*, a classe principal da Semântica de Ações Orientada

a Objetos.

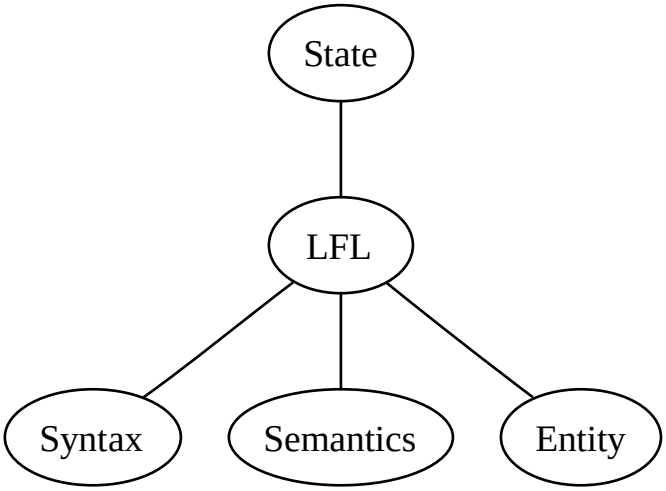

Figura 2.2: Estrutura base da LFL

A primeira versão da LFL foi desenvolvida por Araújo em [Ara04] e aborda apenas o nodo *Semantics*, deixando como sugestão de trabalho o desenvolvimento de uma segunda versão da biblioteca, a qual abordaria os nodos *Syntax* e *Entity*.

Na Semântica de Ações, as funções semânticas definem o significado das frases de uma linguagem. Na LFL, as classes que definem a semântica dos elementos de uma linguagem de programação estão agrupadas no nodo *Semantics*, representado na Figura 2.3.

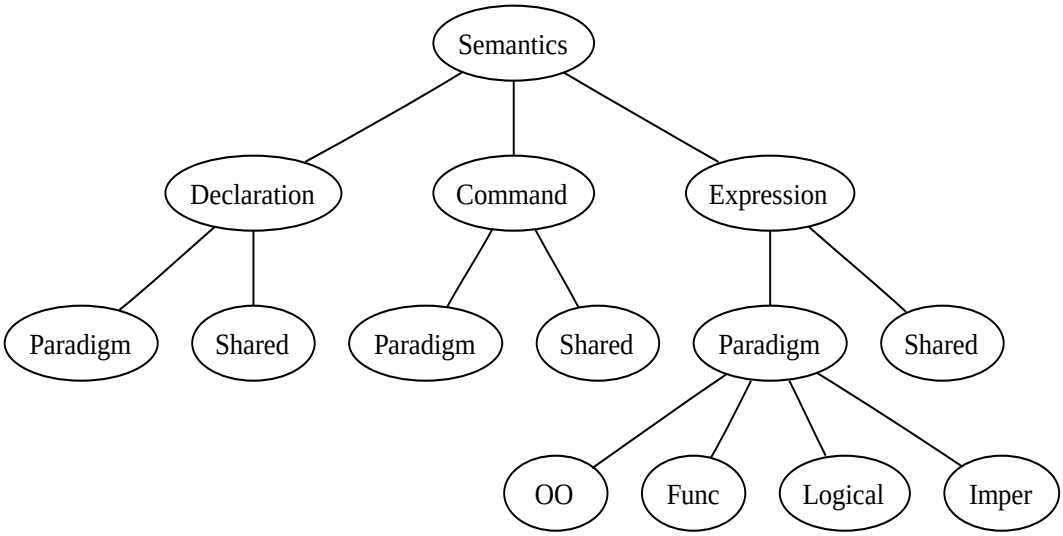

Figura 2.3: Estrutura do nodo Semantics

O nodo *Semantics* é subdivido em: *Declaration*, *Command* e *Expression*. Esta ra-

mificação foi usada para representar os diversos comportamentos que podem existir em uma linguagem de programação. Os programas que modificam *bindings* são representados em *Declaration*; *Command* representa aqueles que mexem com o fluxo de controle e atribuições; o retorno de valores é descrito em *Expression*.

Depois dessa primeira divisão, cada um dos três nodos é subdividido em outros dois: *Paradigm* e *Shared*. O primeiro é responsável por representar as classes que contém características de um paradigma de linguagem de programação específico, por isso foi subdivido de forma que representasse os seguintes paradigmas: orientado a objetos (*00*), funcional (*Func*), lógico (*Logical*) e imperativo (*Imper*). O segundo nodo contém as classes que são comuns a vários paradigmas de linguagens.

A classe Constant, exemplificada a seguir, expressa a semântica para a declaração de constantes em uma linguagem imperativa. Observe que esta classe não possui a parte sintática (syntax). Isto é uma característica de todas as classes que definem a semântica de programas na LFL, pois essas classes se preocupam apenas em expressar o comportamento de programas, sem nenhuma referência a sintaxe deles.

```
Class Constant ≪ Identifier, Expression
    implementing < evaluate _ : Expression → Action > ≫
    locating LFL.Semantics.Declaration.Paradig.Imper
    using E:Expression, I:Identifier
    semantics:
        evaluate-constant(E, I) =
            evaluate ⟦ E ⟧ then bind I to the value
End Class
```

Existem algumas diferenças entre a notação da LFL e a sua equivalente na Semântica de Ações Orientada a Objetos. A principal delas é a adição de parâmetros genéricos na notação usada na LFL, para facilitar o reúso. No exemplo acima as classes Identifier e Expression são dois parâmetros recebidos pela classe Constant. A definição do exemplo não faz nenhuma restrição à primeira classe, porém para a segunda é imprescindível a existência do método evaluate, o qual mapeia uma expressão para uma ação. Este método é considerado o terceiro parâmetro recebido pela classe exemplificada.

A diretiva locating é responsável por indicar a qual nodo da hierarquia, da LFL, a classe pertence. O método evaluate-constant é definido para usar parâmetros mais genéricos, ao

invés das árvores sintáticas usadas na Semântica de Ações Orientada a Objetos.

2.4.2 Sintaxe

Algumas alterações foram efetuadas na notação da Semântica de Ações Orientada a Objetos para que fosse possível adequar as definições de classes no estilo da LFL. A seguir apresentaremos as modificações efetuadas:

(1) Class-Module =
⟦ "Class" Class-Name "≪" Parameter-Class "≫" Class-Body "End-Class" ⟧

Em (1) é redefinida a declaração de uma classe, a qual pode ser parametrizada. Os parâmetros podem ser classes ou métodos e eles são usados na semântica da classe que é definida.

Definimos a declaração de uma classe pelo token Class. Em seguida especificamos o nome da classe (Class-Name) e o token "≪" inicia a parametrização dela, a qual é especificada por Parameter-Class e encerrada por "≫". Após a especificação do corpo (Class-Body), a definição da classe é encerrada pelo token End-Class

(2) Class-Body =
⟦ <"extending" Base-Class-Name>?
<"using" Objects-Declaration>?
<"locating" Structure-Locate>?
<"including" Structure-Locate>?
<Class-Definition >? ⟧

Novas diretivas aparecem na descrição do corpo da classe (2), são elas:

- locating - determina a localização da classe, que está sendo definida, na estrutura organizacional da LFL;
- including - disponibiliza para a classe atual os métodos das classes de um determinado nodo da estrutura da LFL.

Note que ambas as novas diretivas estão relacionadas à estrutura organizacional da LFL, a qual é representada pelo token Structure-Locate.

(3) Object-Declaration =
⟦ Identifier ":" <Structure-Locate"."\>? Class-Name ⟧ |
⟦ Identifier ":" <Structure-Locate"."\>? Class-Name "≪" Parameter-Class "≫" ⟧

Em (3) é redefinida a declaração de um objeto, o qual é a referência a uma classe instânciada. Existem duas formas de declarar um objeto:

- O objeto instanciado é representado por um identificador seguido de ":", opcionalmente pode-se definir a qual estrutura a classe pertence, por meio do token "." e finalmente a classe que está sendo instanciada é especificada;
- A segunda maneira de declaração é similar à primeira. Entretanto, pode-se adicionar parâmetros à classe instanciada. Os parâmetros podem ser outras classes ou métodos dessas classes referenciadas como parâmetros. Tais parâmetros são especificados entre os tokens "≪" e "≫".

(4) Parameter-Class =
⟦ Class-Name ⟧ |
⟦ Class-Name "implementing" "≪" Method-Name "≫" ⟧ |
⟦ Parameter-Class "," Parameter-Class ⟧

Um parâmetro (4), especificado na declaração de uma determinada classe, pode ser apenas o nome de uma classe, uma classe incluindo um dos seus métodos ou uma seqüência de parâmetros.

É a partir da diretiva implementing que um método é passado como parâmetro. Tal método (Method-Name) deve ser declarado entre os tokens "<" e ">".

(5) Structure-Locate =
⟦ Node-Name ⟧ | ⟦ Node-Name "." Structure-Locate ⟧

Como foi apresentado na seção anterior, a estrutura organizacional da LFL é representada em forma de árvore. Para acessar uma determinada classe dessa estrutura, é preciso determinar o seu caminho completo (5). O nodo raiz é indicado por LFL e os demais nodos são acessados por meio do token ".", o qual deve ser usado sempre que for preciso acessar um subnível da árvore.

(6) Semantic-Equation =
⟦ Function-Name "[[" syntax-tree "]]" "=" Action ⟧ |
⟦ Function-Name "(" ⟨ data ⟩? ")" "=" Action ⟧

Em (6) temos a redefinição de uma equação semântica, a qual pode ser um nome de função associado a uma determinada árvore sintática ou pode ser um método que recebe como parâmetro um determinado dado (data), o qual é usado na semântica definida pelo método.

Uma ação sempre será gerada para o método e a definição dele é determinada por um nome seguido de parênteses ("(" ... ")"), os quais são responsáveis por especificar os parâmetros deste método.

(7) Action = ... |
⟦ Object-Name "." Method-Name "(" ⟨ data ⟩? ")" ⟧

É adicionada a chamada a métodos externos pela modificação (7), dessa forma é definido como acessar o método de uma classe. A associação do nome do objeto (Object-Name) ao método (Method-Name) é feita pelo uso do token "." e é necessária para que seja possível a utilização do mesmo. Entre parênteses ("(" ... ")") são especificados os parâmetros do método, os quais podem ser um tipo de dado (data).

A seguir apresentaremos um exemplo que mostra a construção de uma classe pertencente à LFL. O exemplo a seguir é referente a uma classe para a avaliação de um identificador:

```
Class IdentifierEvaluate ≪ Identifier ≫
    locating LFL.Semantics.Expression.Shared
    using I:Identifier
    semantics:
        evaluate-identifier( I ) =
            | give the value bound to I
            or
            | give the value stored in the cell bound to I
End-Class
```

A diretiva locating indica a localização da classe IdentifierEvaluate na estrutura da LFL. A classe definida no exemplo recebe uma outra classe (Identifier) como parâmetro e um objeto deste parâmetro é instanciado. Se o identificador é uma constante, então o valor

associado a ele é retornado e no caso do identificador ser uma variável, o valor armazenado na posição de memória correspondente é retornado em forma de dado transitório (*transient*).

Para deixar mais claro os conceitos da LFL, apresentaremos a seguir um exemplo de uso da classe IdentifierEvaluate.

```
Class Id
    including LFL.Semantics.Expression.Shared
    using objId:IdentifierEvaluate ≪ Id ≫
    syntax:
        Id ::= letter [ letter | digit ]*
    semantics:
        eval ⟦ Id ⟧ =
            objId.evaluate-identifier( Id )
End-Class
```

A especificação acima define uma sintaxe específica para a avaliação de identificadores. Essa classe é um exemplo de especificação de linguagem de programação por meio da LFL. Observe que a sintaxe dos identificadores foi especificada na classe e a sua semântica foi definida de acordo com a biblioteca de classes. A diretiva including é usada para disponibilizar os métodos das classes de uma determinada parte da estrutura da biblioteca e a partir de using são criados os objetos das classes especificadas por including e então é possível acessar os seus métodos.

A definição completa da notação da LFL, bem como as classes disponibilizadas pela sua primeira versão estão descritas em [Ara04].

2.5 Um enfoque construtivo

A maioria das abordagens para a especificação da semântica formal de linguagens são baseadas na suposição de que cada linguagem é definida separada e independentemente. Dessa forma, uma parte da especificação que define um determinado construtor pode depender dos outros construtores inclusos na linguagem. Construtores comuns devem ser definidos para cada linguagem [Mos05].

Na abordagem construtiva, proposta por Mosses, a semântica de cada construtor é

definida separada e independentemente, o conjunto de construtores não é mínimo, e a definição de uma linguagem completa não precisa incluir todos os construtores disponíveis.

A semântica de uma linguagem completa é obtida usando a derivação de vários construtores. Veremos a seguir a definição de construtor, bem como os seus vários tipos e como usá-los em conjunto com a Semântica de Ações.

2.5.1 Construtores

A abordagem proposta por Mosses envolve mapear construtores de linguagens concretas para combinações de construtores básicos abstratos, onde um construtor é a especificação formal de uma determinada característica de uma linguagem. Primeiramente veremos a diferença entre construtores abstratos e concretos. Logo após estudaremos como distinguir construtores básicos de construtores derivados. Em seguida introduziremos os *sorts* básicos dos construtores e como definir construtores básicos individuais. Por fim será apresentada a tradução de construtores concretos para construtores básicos abstratos.

Construtores abstratos e concretos

A sintaxe concreta do while pode ser definida por "while exp do cmd" ou "while(exp) cmd", onde exp é uma condição, representada por uma expressão e cmd é o corpo do laço de repetição, representado por um comando.

A definição por meio da sintaxe concreta inibe a ambigüidade no parsing de um determinado texto de um construtor, enquanto a sua correspondente para a sintaxe abstrata apenas distingue um while de todos os outros construtores, os quais também usam uma expressão e um comando como componentes, por exemplo: if-then.

Na abordagem construtiva é adotada uma notação prefixa independente de linguagem, ou seja, "cond-loop exp cmd" ou "cond-loop(exp, cmd)" representam uma notação particular de um laço condicional.

Construtores básicos e derivados

Uma grande variedade de construtores é encontrada nas linguagens de programação desenvolvidas. Alguns desses construtores são comuns e possuem a mesma interpretação, por isso na abordagem construtiva são denominados construtores básicos. Outros construtores são inclusos em determinadas linguagens, ou possuem comportamentos específicos dependendo da linguagem em que são implementados, por isso são chamados de construtores derivados.

Como exemplo podemos citar as três variações do while:

- Os valores possíveis da condição são booleanos ou os inteiros 0 (zero) e 1 (um);
- A execução de um construtor break no corpo do while interrompe a sua execução;
- A execução de um construtor continue no corpo do while interrompe a iteração atual.

A primeira variação é considerada básica, pois trata de valores booleanos e inteiros, os quais estão presentes na maioria das linguagens. A interrupção de execução não é considerada básica, por isso as demais variações são classificadas como derivadas.

Sorts básicos dos construtores

Veremos agora como os construtores abstratos são classificados. Seguindo o padrão da sintaxe abstrata, os construtores básicos abstratos de programas devem ser classificados de acordo com o *sort* de dados que eles computam. Esses *sorts* de dados podem ser: números, valores verdade, strings, listas, funções, entre outros.

Para qualquer *sort* de dado T, seja T-Cmp o *sort* dos construtores que computam dados do *sort* T, deve-se considerar elementos de dados como construtores abstratos, ou seja, deve-se incluir cada *sort* de dado T em T-Cmp. As seguintes instâncias fornecem a classificação apropriada de alguns construtores básicos:

Exp = Val-Cmp - Expressões são construtores que computam valores de diversos *sorts*. Val é o *sort* de todos os valores das expressões, e Exp, o *sort* das expressões;

Cmd = ()-Cmp - Comandos são construtores que possuem efeitos, mas não computam dado algum. Cmd é o *sort* dos comandos;

Dec = Env-Cmp - Declarações são construtores que alteram ambientes. Env é o *sort* dos ambientes (*environments*) e Dec o *sort* das declarações;

Var = Loc-Cmp - Nas linguagens imperativas temos as variáveis, as quais são construtores que determinam valores armazenados em posições de memória. Loc é o *sort* das localizações para valores e Var o *sort* das variáveis.

Construtores básicos individuais

Acabamos de introduzir alguns *sorts* básicos para a classificação dos construtores. Veremos agora alguns exemplos de construtores básicos individuais, os quais são declarados segundo a notação *Meta-Notation*:

$$s ::= c(s_1,\ldots,s_n)$$

onde, o nome de um construtor é especificado por c, s determina o *sort* do construtor, o qual possui componentes dos *sorts* $s_1,\ldots,s_n$.

Os nomes dos construtores são escritos em letras minúsculas, enquanto os nomes do *sorts* têm a primeira letra maiúscula e as demais minúsculas. Variáveis são escritas com letras maiúsculas. O autor ressalta que os nomes usados para os exemplos dos construtores básicos não estão otimizados e serve apenas como base para demonstrar a abordagem, por isso não se trata a ambigüidade no estudo, sendo assim um possível trabalho futuro.

A seguir estão descritos alguns exemplos, de construtores básicos individuais, apresentados por Mosses em [Mos05]:

Laços de Repetição

Cmd ::= cond-loop(Bool-Exp,Cmd)

O construtor acima representa o comando while com uma condição booleana. Este construtor é considerado básico. O próximo exemplo faz referência a um construtor com comandos separados para a execução antes e depois do teste da condição:

Cmd ::= cond-loop(Cmd,Bool-Exp,Cmd)

O primeiro construtor poderia ter sido definido a partir do segundo, desde que o primeiro comando fosse nulo. O segundo construtor representa o comando for e é considerado derivado porque pode ser facilmente definido na forma mais simples do comando while.

Condicionais

T-Cmp ::= cond(Bool-Exp,T-Cmp,T-Cmp)

Onde T representa todos os *sorts* de dados. Quando T = Val temos expressões condicionais e quando T é vazio temos comandos condicionais.

Dados

T-Cmp ::= T

Os elementos de qualquer *sort* T são considerados construtores básicos. Por isso, os valores booleanos true e false do *sort* de dado Bool são construtores básicos abstratos do *sort* Bool-Exp. Quando T é vazio indica que nenhum dado é computado:

()-Cmp ::= ()

Expressões tipadas

T-Exp ::= typed(Exp,T-Type)

Para cada *subsort* T de Val existe um elemento do *sort* T-Type, ou seja, para o *sort* Bool existe uma entidade de *sort* Bool-Type.

Seqüência

Cmd ::= seq(Cmd,Cmd)

A seqüência de comandos é um construtor básico e a sua ordem de execução é da esquerda para a direita.

Efeitos colaterais

Cmd ::= effect(Exp)

Este construtor básico determina um comando a partir de uma expressão.

Binding

Dec ::= bind-val(Id,BVal-Cmp)

Após computar um valor *bindable*, o construtor acima computa o ambiente que representa o *binding* do identificador para o valor.

De construtores concretos para construtores básicos abstratos

A principal característica da abordagem construtiva é a tradução de construtores de linguagens concretas para a combinação de construtores básicos abstratos. Quando a tradução de cada construtor é definida, a semântica formal dos construtores básicos determina a semântica formal dos construtores concretos [Mos05].

Veremos agora como alguns construtores de linguagens imperativas podem ser traduzidos para combinações de construtores básicos abstratos. O autor adverte que a nova abordagem ainda não adota um formalismo específico e por isso ele apenas demonstra um esqueleto da tradução pretendida.

Expressões condicionais

Em uma expressão if-then-else a condição pode ser restrita apenas a valores booleanos por meio de expressões tipadas. Sendo assim, possui a seguinte forma: cond(typed(-,bool),-,-).

Laços de repetição

Em um laço de repetição, onde o corpo do comando também é uma expressão, o valor de verificação é descartado no fim da execução, ou seja, o valor computado no término de um loop[2] é um valor nulo. Por isso a sua tradução possui a seguinte forma: seq(cond-loop(typed(-,bool),effect(-)),null).

[2]Termo em inglês usado para se referir a laços de repetição

Bindings

A amarração de um identificador a um valor, em uma linguagem imperativa, pode ser traduzida diretamente para o seguinte construtor básico abstrato: bind-val(-,-).

2.5.2 Semântica de Ações usando construtores

O objetivo de usar a Semântica de Ações em conjunto com a abordagem construtiva é proporcionar a definição semântica, independente, de determinados construtores básicos abstratos. A seguir, apresentamos alguns exemplos (originários de [Mos05]) da Semântica de Ações Construtiva:

Computações

action : T-Cmp → Action & using () & giving T

O exemplo acima representa a execução de uma ação.

Construtores de dados

T-Cmp ::= T

action T = give T

Neste exemplo é definido o construtor para retorno de valores, ou seja, se T é um número inteiro então ele é avaliado e retornado.

Seqüência de comandos

Cmd ::= seq(Cmd,Cmd)

action seq(Cmd1, Cmd2) = action Cmd1 and-then action Cmd2

A seqüência de comandos é expressa no exemplo apresentado acima, sendo que Cmd2 é executado apenas após a execução completa de Cmd1.

Condicionais

T-Cmp ::= cond(Bool-Exp,T-Cmp,T-Cmp)

Val ::= Bool

```
action cond(Bool-Exp,T-Cmp1,T-Cmp2) =
    action Bool-Exp then
    maybe check the bool and-then action T-Cmp1 else action T-Cmp2
```

O construtor para expressões condicionais é exemplificado em acima, onde uma expressão booleana é avaliada e a ação T-Cmp1 é computada caso Bool-Exp seja satisfatório; caso contrário a ação T-Cmp2 é computada.

Laços de repetição

```
Cmd ::= cond-loop(Bool-Exp,Cmd)

Val ::= Bool

    action cond-loop(Bool-Exp, Cmd) =
        unfolding
        ( action Bool-Exp then maybe check the bool and-then
        action Cmd and-then unfold else skip )
```

O último exemplo apresenta o construtor de um loop de tal forma que enquanto Bool-Exp é verdadeiro o comando Cmd é executado e o laço de repetição continua sendo executado por meio de unfold; mas quando a condição é falsa o comando skip é executado para que o loop seja encerrado.

2.6 Considerações finais

Neste capítulo apresentamos uma breve introdução à Semântica Denotacional [Sch86] e à Semântica Operacional [Win93], incluindo a Semântica Operacional Estrutural [Plo81] e a Semântica Operacional Estrutural Modular [Mos04]. Por essas abordagens terem uma notação matemática, são consideradas difíceis de ler.

Em seguida apresentamos a Semântica de Ações [Mos92], uma abordagem baseada na Semântica Denotacional e na Semântica Operacional. A qual foi desenvolvida com o intuito de facilitar a leitura de especificações semânticas. Contudo, a Semântica de Ações não provê componentes sintáticos para a criação de módulos e/ou bibliotecas [Gay02]. A especificação semântica é dada por um único arquivo separado em duas partes: sintaxe e semântica.

Vimos três abordagens propostas para solucionar o problema da modularização de código em especificações que usam Semântica de Ações, são elas: Semântica de Ações baseada em módulos [DM01], Semântica de Ações baseada em componentes [MM01] e Semântica de Ações Orientada a Objetos [Car02]. A primeira adicionou suporte à definição de módulos na Semântica de Ações, enquanto na segunda é possível separar uma especificação em Semântica de Ações usando componentes.

Conceitos da orientação a objetos foram introduzidos na Semântica de Ações e assim surgiu a Semântica de Ações Orientada a Objetos. Usando tal abordagem é possível criar uma especificação hierarquizada, a qual é dividida em classes. Entretanto, a reutilização de código ainda era baseada na cópia de código já existente. Por isso, apresentamos uma biblioteca de classes chamada LFL [Ara04]. A qual foi criada usando a abordagem orientada a objetos. A LFL agrega classes em um repositório, as quais podem ser instanciadas e seus métodos podem ser usados em vários projetos relacionados a especificação formal de linguagens de programação.

Por fim, vimos a abordagem construtiva [Mos05]. Trata-se de uma idéia que pode ser incorporada a diversos formalismos. Seu autor tem usado em conjunto com a Semântica Operacional Estrutural Modular e com a Semântica de Ações. Sua principal idéia é usar construtores abstratos para especificar conceitos que são semelhantes em várias linguagens. Esses construtores são definidos separadamente e por meio de uma sintaxe independente de linguagem. A especificação de uma linguagem completa é dada pela combinação de quantos construtores forem necessários.

A seguir veremos que a LFL e a abordagem construtiva possuem bastantes características em comum. Também veremos que a abordagem construtiva pode ser incorporada na Semântica de Ações Orientada a Objetos.

CAPÍTULO 3

USANDO MAUDE

Maude [CDE+05] é uma linguagem reflexiva de alto nível e a sua implementação é um sistema de alta performance que suporta especificações tanto em Lógica Equacional [BJM00] quanto em Lógica de Reescrita [CDE+01]. Apresentaremos uma breve introdução ao uso de Maude, a descrição completa da linguagem pode ser obtida em [CDE+05].

A versão 2.2 do interpretador Maude, implementada em C++, serve como ferramenta formal para a criação de ambientes executáveis para diferentes lógicas, provadores de teoremas, linguagens e modelos de computação [Dur99, MOM93, dOB01, dR05, VMO00, MB04].

Em Maude as especificações e códigos podem ser estruturados usando uma hierarquia de módulos. Tais módulos podem ser módulos de sistema (*system modules*) e módulos funcionais (*functional modules*). A sintaxe para a definição de módulos de sistema é `mod n is D endm`, enquanto para módulos funcionais é `fmod n in D endm`. Em ambos os casos, o nome do módulo é dado por `n` e as declarações por `D`.

As hierarquias de módulos são construídas por meio da inclusão de módulos. A forma mais geral de inclusão é dada pela diretiva `including`, onde *junk* e *confusion* são permitidos, ou seja, ao importarmos `BOOL` em um módulo `TESTE` usando `including`, novas constantes do *sort* `Bool` podem ser criadas (*junk*) e outros significados também podem ser adicionados ao módulo `Bool` (*confusion*). Já a diretiva `protecting` evita tanto *junk* como *confusion*, dessa forma não é possível incluir novas constantes para um determinado *sort*, assim como novos significados também não podem ser adicionados. A forma mais simples de inclusão é dada pela diretiva `extending`, onde *junk* é permitido e *confusion* não é permitido.

Os *sorts* são declarados a partir do uso da diretiva `sort`. As relações entre sorts e subsorts são definidas usando `subsort`. Operadores são definidos pela diretiva `op`,

seguindo a seguinte sintaxe:

```
op o : w -> k [A] .
```

onde o nome do operador é representado pelo símbolo `o`. Os sorts do domínio da operação são representados por `w` (`w = w1 ... wn`) e a imagem é representada por `k`. Atributos equacionais podem ser especificados pela sintaxe `[A]`: onde `assoc` define operadores associativos e `comm` comutativos, por exemplo. Operadores que possuem os mesmos sorts de domínio e imagem podem ser declarados pela variante `ops`.

Em Maude podemos definir metavariáveis antes de usá-las em equações ou regras. Elas são definidas usando a seguinte sintaxe:

```
var v : s .
```

onde o nome da metavariável é definido por `v` e `s` define o seu sort. Várias metavariáveis, de mesmo sort, podem ser definidas usando a diretiva `vars`.

As metavariáveis são usadas em equações e regras. Uma equação incondicional é definida com a seguinte sintaxe:

```
eq [L] : t = t' .
```

onde `[L]` define o rótulo da equação, mas é opcional, e `t` e `t'` são termos do mesmo sort e significa que quando ocorrer `t`, então `t'` deve ser processado.

De maneira similar as equações, regras incondicionais são declaradas usando a sintaxe abaixo:

```
rl [L] : t => t'.
```

novamente, `[L]` opcionalmente define o rótulo da regra, `t` e `t'` são os termos da regra declarada.

Para que o uso de Maude fique mais claro, a seguir mostraremos um exemplo de definição de uma simples calculadora:

Exemplo 3.0.1 (Especificando uma simples calculadora em Maude)

```
fmod CALC is
 protecting INT .

 sort Exp .
```

```
 subsort Int < Exp .

 var E1 : Exp .
 var E2 : Exp .
 var N : Int .

 op [[_]] : Exp -> Int .
 op [[_mais_]] : Exp Exp -> Exp .
 op [[_menos_]] : Exp Exp -> Exp .
 op [[_vezes_]] : Exp Exp -> Exp .
 op [[_div_]] : Exp Exp -> Exp .

 eq [[N]] = N .
 eq [[E1 mais E2]] = [[E1]] + [[E2]] .
 eq [[E1 menos E2]] = [[E1]] - [[E2]] .
 eq [[E1 vezes E2]] = [[E1]] * [[E2]] .
 eq [[E1 div E2]] = [[E1]] quo [[E2]] .
endfm
```

No exemplo 3.0.1 o módulo funcional `CALC` é definido. O módulo `INT` foi incluso para que fosse possível usar o sort `Int`, por ele definido. Usando a diretiva `sort` introduzimos o sort `Exp`, o qual representará as expressões que a calculadora resolve. Os inteiros são definidos como subsorts de `Exp` a partir do uso de `subsort`. As metavariáveis `E1`, `E2` e `N` são definidas, sendo que as duas primeiras são expressões e a última é um número inteiro. As operações de soma, subtração, multiplicação e divisão foram definidas, e a primeira operação definida diz que o resultado final de uma expressão será um número inteiro. As equações aplicam a operação desejada, de acordo com as metavariáveis e operadores passados. Repare que a sintaxe da calculadora é `[[Exp]]`, onde `Exp` pode ser `1 mais 1`, por exemplo. Os *underscores* (`_`) usados nas definições das operações representam os sorts que serão "casados" com a sintaxe criada para processar as equações.

Para usarmos a calculadora devemos criar um arquivo com a especificação do exemplo e carregá-lo no interpretador de Maude:

```
Maude> load calc.maude
```

Após carregado o arquivo, uma operação pode ser executada usando o comando `rewrite`:

```
Maude> rew [[1 mais [[1 mais 1]] ]] .
rewrite in CALC : [[1 mais [[1 mais 1]]]] .
```

```
rewrites: 8 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 3
```

No exemplo acima processamos a seguinte expressão: $1 + (1 + 1)$. O comando `rew` é uma abreviação de `rewrite`.

Até agora vimos que as equações e regras podem ser não condicionais. Entretanto, Maude suporta a definição de equações e regras condicionais; elas são definidas a partir das respectivas sintaxes:

equações condicionais:

```
ceq t = t' if C .
```

regras condicionais:

```
crl t => t' if C .
```

A condição `C` pode ser uma das seguintes opções:

- equações comuns. Quando `t = t'` é satisfeita se e somente se as formas canônicas de `t` e `t'` são igual *modulo* aos atributos especificados nos operadores de `t` e `t'`, como por exemplo associatividade e comutatividade.

- equações booleanas. Por exemplo, `t = true`. Entre as operações booleanas destacamos as de igualdade (`_==_`) e não igualdade (`_=/=_`).

- *matching equations* [CDE+05]. Também são equações comuns, porém são usadas para instanciar novas metavariáveis casando o termo da esquerda com o da direita, por isso em português são chamadas de "equações de casamento". Sua forma é `t := t'`.

- reescritas. Considerando `t => t'`, sendo que `t` e `t'` são termos de qualquer sort, temos a condição satisfeita quando existe zero ou mais reescritas de `t` em `t'`.

A condição `C` pode ser apenas uma ou uma conjunção de condições combinadas com o operador `_/\_`. Reescritas condicionais só devem ser usadas em regras condicionais.

Vimos uma breve introdução do uso de Maude [CDE+05], baseado em [dR05]. Maude deu o suporte necessário para a criação da ferramenta Maude MSOS Tool [CB05], também chamada de MMT, a qual veremos a seguir.

3.1 Maude MSOS Tool

Nesta seção vamos apresentar Maude MSOS Tool (MMT), um ambiente para a programação de especificações em Semântica Operacional Estrutural Modular (Modular SOS). MMT é uma ferramenta formal implementada como uma extensão conservativa de Full Maude que compila especificações MSOS para Lógica de Reescrita [dR05].

A sintaxe adotada pela MMT é baseada na sintaxe MSDF (*Modular SOS Definition Formalism*), usada em especificações MSOS. As duas linguagens são similares, o que garante que as especificações usando a MMT são mais próximas possíveis das especificações em MSDF. No entanto, pequenas diferenças existem devido às peculiaridades do parser de Maude.

A MMT é uma ferramenta muito útil para a definição formal de linguagens de programação, visto que é possível definir a sintaxe abstrata da linguagem usando a notação **BNF** e a semântica da linguagem é dada por *regras de transição* com *rótulos* contendo os componentes semânticos necessários. Também é possível organizar a especificação em módulos, o que garante a construção de uma especificação modular, sendo assim possui grande reusabilidade e bibliotecas podem ser construídas.

3.1.1 Sintaxe MSDF

A seguir apresentaremos de forma resumida a sintaxe da linguagem de especificação MSDF, implementada na MMT. Cada seção apresentará alguns dos construtores implementados por meio de exemplos de uso.

3.1.1.1 Módulos

A especificação de módulos em MSDF começa com a diretiva `msos` e termina com `sosm`. O nome de um módulo MSDF é um identificador aceito por Maude (para mais informações ver a Seção 2 de [CB05]).

Outros módulos podem ser inclusos a partir do uso do construtor `see`. Além de uma lista de módulos poder ser inclusa, com os módulos separados por vírgula (`,`), múltiplas

linhas de inclusão também podem ser definidas, no entanto a MMT se encarrega de transformar múltiplas linhas de inclusão em apenas uma linha de inclusão de módulo.

O resto da especificação do módulo é uma seqüência de declarações MSDF, a qual será detalhada na próxima seção.

A seguir definimos o módulo MSDF `LANG`, no qual exemplificaremos a inclusão de módulos definida anteriormente. Note que as inclusões são apenas definições de características de uma linguagem fictícia.

```
msos LANG is
 see DECLARATIONS, COMMANDS .
 see EXPRESSIONS .
sosm
```

3.1.1.2 Definições de tipos de dados

Em MSDF, a sintaxe abstrata de linguagens de programação é especificada em notação **BNF** estendida, a qual é na verdade uma definição algébrica de tipos de dados. Na MMT funções *mixfix* são usadas, onde os não-terminais são os argumentos e os terminais são os nomes das funções em formato *mixfix*. Isso se deve ao fato de que não-terminais são conjuntos, seqüências de não-terminais e terminais. Uma função que contém apenas um terminal é considerada uma *constante*.

Um identificador deve ter a primeira letra em caixa alta (ex: `Integer`). Os identificadores são chamados de *conjuntos primitivos* e devem conter apenas letras, por causa das restrições existentes em relação a formação de metavariáveis. No entanto, um identificador também pode ser considerado como um *conjunto derivado*, onde `Integer*` representa o conjunto das seqüências de elementos e `Integer+` o conjunto das seqüências não-vazias de elementos. Esses conjuntos são automaticamente derivados dos seus respectivos conjuntos primitivos.

Um novo conjunto, ou uma inclusão de conjuntos, ou uma declaração de função são aceitos pela **BNF** implementada na MMT. As declarações de funções dependem do que é especificado do lado direito da regra. No lado direito podemos ter apenas uma inclusão de conjunto (ex: `Exp ::= Value .`), ou então uma seqüência de identificadores, interpre-

tada como declaração de função *mixfix* (ex: `Exp ::= if Exp then Exp else Exp .`). A forma prefixada nas funções *mixfix* (ex: `Sys ::= parallel (Cmd, Cmd)`) também é aceita. As funções *mixfix* são basicamente compostas de operadores em caixa baixa e conjuntos. A correta forma de usar atributos opcionais é descrita na seção 3.2 de [CB05].

Até a versão 2.2, dois conjuntos são previamente definidos pela MMT: inteiros (`Int`) e booleanos (`Boolean`). Os valores verdadeiro e falso são representados pelas constantes `tt` e `ff`, respectivamente.

Usando a MMT também é possível definir conjuntos parametrizados, os quais são obtidos a partir da aplicação de um modificador sobre um ou dois conjuntos. Uma lista de elementos do conjunto `s` é implementada usando `(s) List`. Um conjunto finito de elementos pertencentes a `s` é representado por `(s) Set` e `(s,k) Map` é o conjunto de mapeamentos finitos dos elementos de `s` para `k`. Os conjuntos parametrizados devem ser usados por meio dos seus *conjuntos equivalentes*.

A seguir apresentamos um exemplo de definição de tipos de dados em MSDF. Expressões, valores, comandos e identificadores de uma linguagem hipotética serão representados pela declaração dos conjuntos: `Exp`, `Value`, `Cmd` e `Id`. Os conjuntos pré-definidos `Int` e `Boolean` são inclusos no conjunto `Value`. As sintaxes abstratas de uma expressão condicional e de um laço de repetição são definidas na forma *mixfix*. O corpo do comando de repetição deverá ser uma seqüência de comandos não-vazia. Na última linha é exemplificada uma equivalência entre os conjuntos `Env` e `(Id, Value) Map`, onde é definido um mapeamento entre identificadores e valores.

```
Exp .  Value .
Cmd .  Id .
Value ::= Int | Boolean .
Exp ::= if Exp then Exp else Exp .
Cmd ::= while Exp do Cmd+ .
Env = (Id, Value) Map .
```

3.1.1.3 Rótulos

Apenas uma declaração de rótulo é permitida em cada módulo MSDF. Um rótulo é uma seqüência de declarações de tipos de campos. Cada campo é representado por um índice

e pelo tipo do componente do índice. Os índices devem ser declarados em caixa baixa.

Os índices dos componentes dos campos dos rótulos podem ser *read-only* (leitura e escrita), *read-write* (escrita e leitura) ou *write-only* (somente escrita). Temos um componente *read-only* quando existe apenas um índice não primalizado:

```
Label = { env : Env, ... } .
```

Quando existe um índice primalizado e não primalizado temos um componente *read-write*. Ambos os componentes devem ser do mesmo tipo, como no exemplo a seguir:

```
Label = { sto : Store, sto' : Store, ... } .
```

Um único índice primalizado define um componente *write-only*. Apenas seqüências de conjuntos primitivos são admitidas nos componentes *write-only*:

```
Label = { output' : Value*, ... } .
```

No caso de várias declarações de rótulos serem especificadas em um mesmo módulo, só a última será considerada. Por isso, uma única declaração de rótulo pode definir vários campos:

```
Label = { env : Env, sto : Store, sto' : Store, output' : Value*, ... } .
```

3.1.1.4 Transições semânticas

As transições MSOS na MSDF da MMT podem ser condicionais ou não condicionais. Em ambos os tipos de transições é possível especificar um rótulo opcional [1].

Uma transição incondicional é uma relação entre termos e pode ser escrita no estilo da semântica *big-step* (`==>`), da semântica *small-step* (`-->`), ou ainda da semântica estática (também `==>`). Essas relações descritas são ternárias e possuem os seguintes componentes: a árvore sintática com valores tipada a ser casada, a expressão do rótulo MSOS e a árvore sintática com o resultado da transição.

[1] Não confundir com os rótulos das transições MSOS, pois este é apenas um rótulo decorativo para nomear as regras especificadas

Uma expressão de rótulo é uma lista não vazia de expressões de campos. Essa lista deve estar envolta entre chaves e cada campo é separado por vírgulas. Cada expressão de campo é um índice do rótulo mais o seu componente ou o resto do rótulo. A metavariável `-` deve ser usada se o resto do rótulo é não observável, caso contrário `...` deve ser usada. Em vez de usar `-{-}->` e `={-}=>` pode-se usar `-->` e `==>`. As metavariáveis `X` e `U` podem ser usadas no lugar de `...` e `-`, respectivamente.

Para facilitar a declaração de metavariáveis sobre conjuntos, elas são declaradas implicitamente na MMT. Por exemplo: `Exp`, `Exp1` e `Exp'` são metavariáveis do conjunto `Exp`. Para maiores informações sobre as metavariáveis sugerimos a leitura da seção 3.4 de [CB05].

A seguir apresentamos um exemplo de transição incondicional. Uma transição incondicional entre uma árvore sintática `(print Value) : Cmd` de valores tipada e o comando `skip` é exemplificada. Onde a semântica do comando `print` é especificada por um componente *write-only*, com uma expressão de rótulo `out' = Value` para modelar a informação produzida por `Value`.

```
(print Value) : Cmd -{out' = Value}-> skip .
```

As transições condicionais podem ser compostas de condições equacionais, onde a condição é satisfeita se a igualdade de dois termos é verificada (ex: `first (Pids') = Int`). Um termo `P` pode ser usado como predicado para abreviar uma condição equacional. Por exemplo, `odd(n)` abrevia `P = true`. Uma instanciação de metavariável também pode ser usada nas condições, por exemplo, `Value := lookup (Id, Env)`. Outro tipo de condição aceito é a transição que possui a mesma sintaxe de transições condicionais.

A especificação *small-step* da semântica de uma expressão condicional, de sintaxe `if Exp then Exp else Exp`, será usada como exemplo de transição condicional. De maneira similar a transições MSOS. Primeiro é definida uma transição que a primeira expressão é avaliada e gera um valor booleano. De acordo com o valor verdade gerado é executada uma das outras duas expressões. Por isso é necessário especificar três regras.

Na primeira transição a metavariável `Exp1` sobre o conjunto `Exp` é computada para gerar o valor booleano. O termo `if Exp1 then Exp2 else Exp3 : Exp` é uma árvore

tipada do tipo `Exp`, enquanto do lado direito da regra é especificado o que acontecerá após a avaliação de `Exp1`. Na parte de cima da transição o termo `Exp1` é avaliado para `Exp'1`. O rótulo `...` é usado tanto na conclusão como na condição e significa que as alterações efetuadas em componentes *read-write* e *write-only* são propagadas.

```
                    Exp1 -{...}-> Exp'1
-- ---------------------------------------------------------------------
if Exp1 then Exp2 else Exp3 : Exp -{...}-> if Exp'1 then Exp2 else Exp3 .
```

Os três traços (`---`) que iniciam um comentário de linha em Maude são usados para deixar a regra MMT o mais parecida possível com especificações MSOS.

Por fim apresentamos as duas outras regras necessárias para concluir a avaliação de uma expressão condicional. Se `Exp1` é avaliado para `tt`, a expressão `Exp2` é computada e caso contrário `Exp3` é computada.

```
if tt then Exp2 else Exp3 : Exp --> Exp2 .
if ff then Exp2 else Exp3 : Exp --> Exp3 .
```

Apresentamos exemplos usando apenas a semântica no estilo *small-step*, exemplos usando outros estilos podem ser obtidos em [CB05].

3.1.1.5 Operações pré-definidas

A MMT implementa algumas funções para serem aplicadas sobre conjuntos derivados e parametrizados. Mais especificamente, são implementadas funções para operar sobre seqüências, listas, mapas e conjuntos.

Por exemplo, uma entrada `s` em uma mapa `k` é criada pela função a seguir:

```
_|->_ : s x k -> (s,k) Map
```

E o tamanho de uma lista `s` é retornado usando a seguinte função:

```
length : (s) List -> Nat
```

Todas as funções disponibilizadas pela MMT são descritas na seção 3.5 de [CB05].

3.1.2 Exemplo de uso

Nesta seção vamos exemplificar o uso da MMT através da especificação de uma linguagem de expressões aritméticas, similar a do exemplo 2.1.8 da seção 2.1.3.2.

O nome da linguagem será `EXEMPLO` e ela será especificada em um módulo MSDF, o qual é definido da seguinte maneira:

```
(msos EXEMPLO is ... sosm)
```

As especificações de módulos na MMT devem sempre estar entre parênteses, isso por causa das alterações efetuadas no `LOOP-MODE` de Maude [dR05]. Em seguida especificamos quais são os conjuntos usados na linguagem. Nesse caso usaremos `Exp` para representar as expressões e `Id` os identificadores.

```
Id .
Exp .

Id ::= x | y .

Exp ::= Int
      | Id
      | Exp mais Exp
      | let Id recebe Exp in Exp .
```

Repare que além de definirmos os conjuntos usados na especificação, também definimos os identificadores que poderão ser usados e a sintaxe abstrata da linguagem usando a notação **BNF**. Na nossa linguagem de exemplo, uma expressão pode ser um número inteiro, um identificador, uma soma de expressões ou ainda uma expressão "`let`". Além dos conjuntos usados nas especificações, devemos definir o rótulo MSOS e os seus componentes, como é descrito a seguir:

```
Env = (Id, Int) Map .

Label = { env : Env, ... } .
```

O ambiente `Env` é definido como um mapa de identificadores para números inteiros e ele é o componente do índice `env`, do rótulo do nosso exemplo. É usado um componente *read-only*, já que é uma característica de expressões "`let`". Em seguida especificamos a

avaliação de um identificador no ambiente:

```
   Int := lookup(Id, Env)
-- ----------------------------
Id : Exp -{env = Env, -}-> Int .
```

Usando a função `lookup`, pré-implementada pela MMT, obtemos o número inteiro que está amarrado ao identificador `Id` no ambiente `Env`. Para processar a soma de expressões serão necessárias três regras:

```
       Exp1 -{...}-> Exp'1
-- ----------------------------------------
Exp1 mais Exp2 : Exp -{...}-> Exp'1 mais Exp2 .

       Exp2 -{...}-> Exp'2
-- ---------------------------------------
Int mais Exp2 : Exp -{...}-> Int mais Exp'2 .

   Int3 := Int1 + Int2
-- ----------------------------
Int1 mais Int2 : Exp --> Int3 .
```

Na primeira regra a expressão `Exp1` é avaliada para um número inteiro. O mesmo acontece com a expressão `Exp2` na segunda regra. Após termos as duas expressões avaliadas damos a soma dos seus inteiros na terceira regra. Nossa expressão "`let`" também é especificada em três regras:

```
               Exp1 -{...}-> Exp'1
-- ------------------------------------------------------------------
let Id recebe Exp1 in Exp2 : Exp -{...}-> let Id recebe Exp'1 in Exp2 .

       Env' := (Id |-> Int1) / Env, Exp2 -{env = Env', ...}-> Exp'2
-- ------------------------------------------------------------------
let Id recebe Int1 in Exp2 : Exp -{env = Env, ...}->
   let Id recebe Int1 in Exp'2 .

let Id recebe Int1 in Int2 : Exp --> Int2 .
```

A primeira regra é usada apenas para avaliar a expressão `Exp1`. Já na segunda regra, o inteiro avaliado pela regra anterior é amarrado ao identificador em um novo ambiente, o qual é usado para avaliar a expressão `Exp2`. Após as duas expressões serem avaliadas, o resultado final é dado pela última regra.

Para executar o exemplo descrito, basta usar o ambiente proporcionado ao carregar a MMT em Maude:

```
$ maude mmt
```

Em seguida devemos carregar o arquivo que contém a especificação:

```
Maude> load exemplo.msdf
rewrites: 100336 in 1000ms cpu (1431ms real) (100336 rewrites/second)
Introduced MSOS module EXEMPLO
```

O comando `rewrite` pode ser usado para testar a especificação executando uma construção pertencente a linguagem. Note que o argumento a ser passado é uma configuração da Semântica de Reescrita Modular, do inglês Modular Rewriting Semantics (MRS) [MB04], usada na implementação da MMT [CB05].

```
Maude> (rew < let x recebe (1 mais 2) in (x mais 4) :::
        'Exp, {env = void} > .)
rewrites: 1856 in 60ms cpu (327ms real) (30933 rewrites/second)
rewrite in EXEMPLO :
  < let x recebe 1 mais 2 in(x mais 4)::: 'Exp,{env = void}>
result Conf :
  < 7 ::: 'Exp,{env = void}>
```

3.1.3 Limitações

A MMT é uma excelente ferramenta para ser usada na especificação formal de linguagens de programação no formalismo MSOS. No entanto, ela possui algumas limitações relacionadas ao próprio Maude, bem como a sua sintaxe MSDF. Devemos conhecer essas limitações para planejar melhor a utilização da ferramenta, por isso apresentamos algumas das limitações nessa seção.

Uma das limitações da MSDF implementada na MMT está na declaração de funções e subconjuntos. Ou seja, as funções devem sempre ser declaradas antes dos subconjuntos, pois caso contrário a ferramenta pode se confundir. Por exemplo, use:

```
Exp ::= sum (Exp, Exp) | Int | Id .
```

Em vez de usar:

```
Exp ::= Int | Id | sum (Exp, Exp) .
```

Uma restrição diretamente relacionada a Maude é o problema de pré-regularidade [CB06]. Isto porque os tipos de dados definidos em MSDF estão sujeitos as mesmas restrições aos tipos de dados definidos em Maude, já que as especificações que usam a **BNF** de MSDF são traduzidas para sorts, subsorts e operações em Maude. Ou seja, um termo deve conter apenas um sort, pois isso seria um requerimento da pré-regularidade. Nesse caso, o exemplo a seguir teria o problema de pré-regularidade:

```
(msos PROBLEMA-PRE-REGULARIDADE is
 Id .
 Exp .
 Value .
 Storable .

 Storable ::= Int .

 Value ::= Int .

 Exp ::= sum (Exp, Exp) | Value | Id .

 Env = (Id, Int) Map .

 Label = {env : Env, ...} .
sosm)
```

Repare que o conjunto dos inteiros (`Int`) é definido como um subconjunto de `Storable` e `Value`. Isto acarreta o problema discutido. Ao executarmos um comando `rewrite` para o termo `Int` um *warning* é gerado:

```
Maude> (rew < 1 ::: 'Exp, {env = void} > .)
Warning: sort declarations for operator _:::_ failed preregularity check
on 5 out of 42 sort tuples. First such tuple is (Int, Qid).
```

Para resolver o problema podemos supor que um `Storable` é um elemento de `Int` e que um `Value` é um elemento de `Storable`:

```
(msos SOLUCAO-PREREGULARIDADE is
 Id .
 Exp .
 Value .
 Storable .

 Storable ::= Int .
```

```
 Value ::= Storable .

 Exp ::= sum (Exp, Exp) | Value | Id .

 Env = (Id, Int) Map .

 Label = {env : Env, ...} .
sosm)
```

A última limitação, aqui reportada, no uso da MMT é o problema de *ad-hoc overloading*. Esse tipo de sobrecarga de operadores e subsorts é permitida em Maude. Porém, requere-se que os sorts nas aridades de duas operações que possuem a mesma forma sintática e os sorts nas coariades devem pertencer ao mesmo componente conexo [CDE+05]. Tudo isso para evitar expressões ambigüas. No exemplo a seguir é mostrado o problema:

```
(msos PROBLEMA-OVERLOADING is
 Id .
 Exp .

 Exp ::= sum (Exp, Exp) | Int | Id .

 Env = (Id, Int) Map .

 Sto = (Int, Int) Map .

 Label = { env : Env, sto : Sto, sto' : Sto, ... } .
sosm)
```

Repare que os mapas `Env` e `Sto` possuem o mesmo domínio, mas a imagem é diferente. Além disso, tanto imagem como domínio são subconjuntos de `Exp`. Essa configuração gera o problema de *ad-hoc overloading* discutido. Veja o erro reportado ao carregarmos o módulo e tentarmos executar o comando `rewrite`:

```
Maude> (rew 1 .)
Warning: <metalevel>: declaration for _|->_ has the same domain kinds as
the declaration on <metalevel> but a different range kind.
Advisory: could not find an operator _+++_ with appropriate domain in
meta-module EXEMPLO when trying to interprete metaterm
'_+++_['_|->_['K:Id,'V:Int],'M:Map`{Id`,Int`}].
```

A primeira solução seria seguir a instrução dada pelo manual de Maude, ou seja, reunir em apenas um componente conexo a imagem das duas operações. A solução é descrita no

módulo SOLUCAO1, o qual é um módulo funcional de Maude que inclui o módulo MMT definido e cria o componente conexo para resolver o problema.

```
(fmod SOLUCAO1 is
 including EXEMPLO .

 sort MajorEntry .
 subsort Entry{Id, Int} Entry{Int, Int} < MajorEntry .
endfm)
```

A segunda solução seria usar funções de coerção, pois podem ser *ad-hoc overloaded* para representar a hierarquia de Exp. Esta solução é mostrada no módulo MSDF SOLUCAO2, o qual nada mais é do que o módulo PROBLEMA-OVERLOADING com o conjunto Exp usando a função de coerção < > aplicada sobre o conjunto dos inteiros. A função de coerção pode ser usada da maneira que o usuário da MMT achar mais conveniente.

```
(msos SOLUCAO2 is
 Id .
 Exp .

 Exp ::= sum (Exp, Exp) | < Int > | Id .

 Env = (Id, Int) Map .

 Sto = (Int, Int) Map .

 Label = { env : Env, sto : Sto, sto' : Sto, ... } .
sosm)
```

A aplicação da função de coerção no conjunto dos inteiros já resolveu o problema, mas caso deseje-se aplicá-la também aos identificadores, para padronizar o uso da mesma função para os valores processados pelo exemplo, seria preciso colocar as especificações em linhas separadas, pois, caso contrário, esbarra-se em um problema de *parsing* da MMT.

```
Exp ::= sum (Exp, Exp) .
Exp ::= < Int > .
Exp ::= < Id > .
```

Diante das duas soluções sugerimos a segunda, pois ao usar a primeira pode-se comprometer a semântica da linguagem que está sendo especificada.

3.2 Considerações finais

Neste capítulo apresentamos Maude [CDE+05], uma linguagem reflexiva e também um sistema que suporta tanto lógica equacional como lógica de reescrita. Maude permite a criação de ambientes executáveis para linguagens de programação, provadores de teoremas, diferentes lógicas e modelos computacionais.

Em seguida apresentamos MMT (*Maude MSOS Tool*) [dR05], um ambiente executável para a Semântica Operacional Estrutural Modular (MSOS) [Mos04]. Usando MMT é possível criar especificações MSOS executáveis. Para isso, foi criada e implementada uma linguagem chamada MSDF (*Modular SOS Definition Formalism*), a qual contempla a especificação de sintaxe abstrata em uma notação BNF estendida e a especificação de transições MSOS em uma representação textual que captura a notação matemática comunmente usada na Semântica Operacional Estrutural Modular.

Devido a implementação e criação da MSDF, as especificações MSOS usando a MMT são muito parecidas com as especificações do formalismo. A ferramenta é bastante útil para a especificação formal usando MSOS e por isso ela foi usada na implementação da Notação de Ações da Semântica de Ações Orientada a Objetos. Também apresentamos algumas limitações da ferramenta, as quais não impediram o processo de desenvolvimento.

CAPÍTULO 4

A LFL VERSÃO 2

Este capítulo trata da atualização da biblioteca de classes para a Semântica de Ações Orientada a Objetos. Primeiramente apresentaremos o que motivou a reestruturação da LFL. Em seguida mostraremos o que muda na definição das classes da biblioteca, bem como na sua estrutura organizacional e sintaxe. Para finalizar, apresentaremos as classes disponibilizadas na nova versão.

4.1 O Problema

Os formalismos usados para descrever a semântica de linguagens de programação vêm sendo modificados para aprimorar a leitura, escrita e reusabilidade das especificações semânticas. Atualmente, já é possível especificar a semântica de uma linguagem independentemente da sua sintaxe.

A Semântica de Ações inerentemente possui boas propriedades de reusabilidade e extensibilidade [MM94]. No entanto, a notação da Semântica de Ações não possui suporte sintático para a definição de blocos componíveis. Dessa forma, não é possível criar bibliotecas para que os seus componentes sejam reutilizados em novas descrições [Gay02].

A Semântica de Ações Orientada a Objetos [Car02] foi desenvolvida para melhorar a questão da modularidade na Semântica de Ações por meio do uso de conceitos da orientação a objetos, como por exemplo: definição de classes e métodos. Já que é possível dividir a especificação em classes, usando o formalismo orientado a objetos, então uma biblioteca de classes, chamada LFL [Ara04], foi desenvolvida para melhorar o reúso de especificações semelhantes em um ou vários projetos diferentes. Contudo, consideramos que alguns novos problemas de leitura e escrita foram introduzidos pela LFL.

Para elucidar o problema, vejamos a especificação da classe Selection da LFL:

```
Class Selection
        ≪ Command implementing < execute _ : Command → Action >
        Expression implementing < evaluate _ : Expression → Action > ≫
    locating LFL.Semantics.Command.Shared
    using E:Expression C1:Command, C2:Command
    semantics:
        execute-if-then(E, C1) =
            evaluate ⟦ E ⟧ then
            | check(the given TruthValue is true) and then execute ⟦ C1 ⟧
            or
            | check(the given TruthValue is false) and then complete
        execute-if-then-else(E, C1, C2) =
            evaluate ⟦ E ⟧ then
            | check(the given TruthValue is true) and then execute ⟦ C1 ⟧
            or
            | check(the given TruthValue is false) and then execute ⟦ C2 ⟧
End Class
```

A classe Selection foi implementada para prover a semântica de um comando de seleção, independentemente da sua sintaxe. Para usar esta classe é necessário passar como parâmetros as classes Command e Expression, as quais devem ser definidas e nomeadas pelo usuário. A partir da diretiva implementing, as funções semânticas implementadas nas classes Command e Expression também são passadas como parâmetros. A localização da classe na estrutura da LFL é definida pela diretiva locating.

A semântica do comando é expressa por qualquer um dos métodos especificados na classe exemplificada: execute-if-then ou execute-if-then-else. A diferença entre os dois métodos é que o primeiro recebe uma expressão e um comando como parâmetros, enquanto o segundo recebe uma expressão e dois comandos como parâmetros.

As expressões e comandos são definidos, na linguagem implementada, pelo usuário e a LFL é capaz de avaliá-los porque as classes que implementam as expressões e comandos são passadas como parâmetros para a classe da LFL. Para que fique mais claro, vejamos como seria a especificação do comando seleção, no estilo if-then-else, usando a LFL:

```
Class Comando
    syntax:
        Com
    semantics:
        executar _ : Com → Action
End Class
```

A classe Comando, funciona como uma classe abstrata e é um típico exemplo de uma classe definida pelo usuário e que deve ser passada para a LFL para sincronizar a biblioteca com a especificação do usuário.

```
Class Selecao
    extending Comando
    using C1:Comando, C2:Comando, E:Expressao,
    objSel:LFL.Semantics.Command.Shared.Selection ≪
        Comando<executar>, Expressao<avaliar> ≫
    syntax:
        Com ::= "if" E "then" C1 "else" C2 "end-if"
    semantics:
        executar ⟦ "if" E "then" C1 "else" C2 "end-if" ⟧ =
            objSel.execute-if-then-else(E, C1, C2)
End Class
```

Na classe Selecao temos uma classe especializada. Ela implementa o comando de seleção usando a classe Selection da LFL. Isso é possível pela instanciação do objeto *objSel*. Repare que na diretiva using, para efetuar a instanciação do objeto é preciso passar como parâmetros as classes Comando e Expressao, bem como os métodos dessas classes, definidos pelo usuário, e é dessa forma que a LFL garante que a especificação semântica será conseguida no uso do método execute-if-then-else.

O advento da LFL traz um conceito interessante para a Semântica de Ações Orientada a Objetos: descrições semânticas independente da sintaxe da linguagem por meio do uso das classes da biblioteca. Por outro lado, o uso da LFL, bem como a passagem de parâmetros, é um tanto obscura e volta problemas de leitura e escrita encontrados em especificações de linguagens de programação. Não obstante, do ponto de vista da orientação a objetos, a renomeação de métodos também é considerada uma solução não elegante.

4.2 A Solução

Para solucionar o problema relatado na seção anterior, propomos a inibição da passagem de parâmetros nas classes da LFL. Isso é possível por meio do uso de *abstrações* de *ações* da Semântica de Ações.

De acordo com a nossa proposta, a classe Sequence (mostrada na seção anterior) seria escrita da seguinte maneira:

```
Class Sequence
    locating LFL.Command.Shared.Sequence
    using: Y1:Yielder, Y2:Yielder, Y3:Yielder
    semantics:
        execute-if-then(Y1, Y2) =
            | enact Y1 then enact Y2 else complete
        execute-if-then-else(Y1,Y2,Y3) =
            | enact Y1 then enact Y2 else enact Y3
End Class
```

Agora, as classes da LFL são mais compactas, pois elas apenas definem a sua própria localização na hierarquia da LFL, os métodos que descrevem um determinado conceito de linguagens de programação e os objetos usados nos métodos especificados.

Isto é possível pelo uso de *abstrações* e da faceta reflexiva. Ao invés de passar classes que especificam os métodos usados nas classes da LFL, cada método implementado na nova biblioteca recebe um determinado número de objetos, os quais são *abstrações* que são computadas de acordo com o comportamento que deseja-se implementar em uma determinada classe da LFL. Veremos a seguir que na verdade, as *abstrações* são encapsuladas por *yielders* e ao usar a *ação* enact é possível executar uma *abstração* de uma *ação* sem precisar de nenhum método definido pelo usuário.

Outra modificação que ressaltamos nesse ponto é o novo uso da diretiva locating. Anteriormente ela especificava o nodo pai da classe que estava sendo definida. Optamos por dar a localização exata da classe na hierarquia, pois no caso de uma implementação da LFL, para obtermos a localização de uma determinada classe seria preciso concatenar o conteúdo de locating com o nome da classe. Mas, se especificarmos a localização exata, basta saber o conteúdo de locating.

Quando as novas classes forem apresentadas, veremos que além de *abstrações* os métodos implementados nas classes da biblioteca também podem receber *tokens* e *sorts*. Tanto *tokens* como *sorts* estão presentes na Semântica de Ações Orientada a Objetos e serão responsáveis por representar os identificadores e os tipos de dados implementados na linguagem do usuário da LFL.

4.2.1 Estrutura Organizacional

Além da inibição de parâmetros, também propomos o uso da LFL apenas para conceitos semânticos. Tomamos essa decisão baseado no fato de que as linguagens de programação, apesar de possuirem sintaxes diferentes podem efetuar computações semelhantes. Por isso, julgamos desnecessária a preocupação da biblioteca com elementos sintáticos. Também é desnecessária a preocupação da biblioteca com entidades semânticas, já que elas são características inerentes da Semântica de Ações.

Isto implica na exclusão dos nodos: *Syntax*, *Semantics* e *Entity*. Já que vamos usar a LFL apenas para representar a semântica de linguagens de programação, então podemos ligar os nodos *Declaration*, *Command* e *Expression* diretamente com o nodo principal da LFL. Para que fique mais claro, apresentamos a estrutura atual da LFL na Figura 4.1:

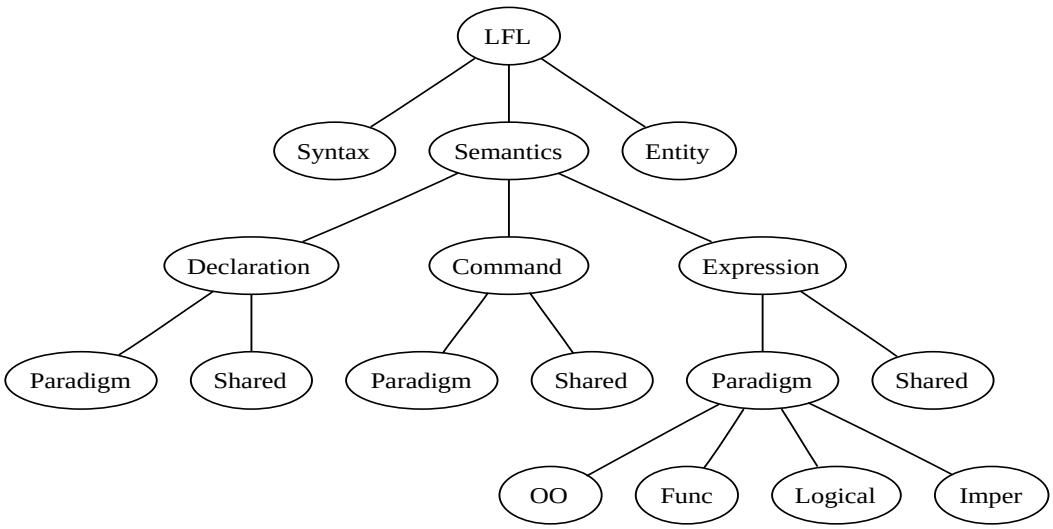

Figura 4.1: Antiga Estrutura da LFL

Seguindo a idéia de especificar a semântica de linguagens independentemente de sintaxe, proposta em [Mos05], mantemos os nodos *Shared* e *Paradigm*, assim como as respectivas subdivisões deles.

Apenas ressaltamos que essa decisão foi tomada devido ao fato de que, assim como na Semântica de Ações Construtiva [Mos05, IM05], também estamos preocupados em separar as especificações semânticas que são comuns em diversas linguagens, esse seria o papel do nodo *Shared*, das características que são particulares, ou seja, aquelas que estão presentes

apenas em algumas linguagens e normalmente estão atreladas a um paradigma específico, por isso o nodo *Paradigm* e os seus subnodos.

Tendo em vista as considerações apresentadas, a nova estrutura da LFL pode ser representada pela Figura 4.2:

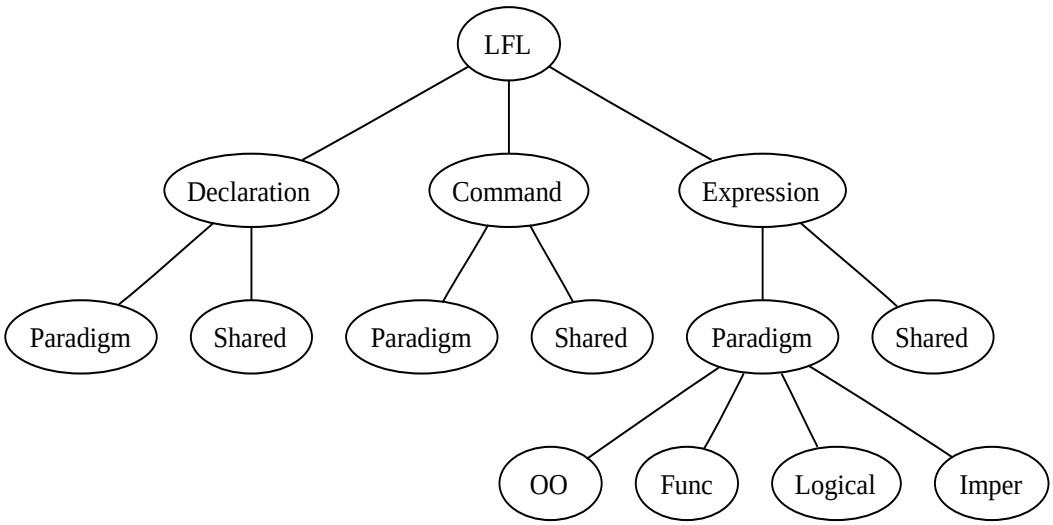

Figura 4.2: Nova Estrutura da LFL

4.2.2 Sintaxe

Para a criação das *abstrações* fizemos a adição de um *yielder* à notação da Semântica de Ações Orientada a Objetos. Essa alteração pode ser conferida a seguir:

```
Yielder =
    ... |
    ⟦ "encapsulate" Action ⟧
```

De fato, a cláusula encapsulate é uma abreviação do seguinte *yielder*:

- closure abstraction of _ : Action → Yielder

Usando a diretiva encapsulate é possível criar as abstrações a serem usadas pelas novas classes da LFL. Note que esta diretiva deve ser usada na especificação de linguagens com *bindings* estáticos. Se a linguagem que está sendo especificada possui *bindings* dinâmicos, então a diretiva closure deve ser eliminada da especificação, ou seja, ao invés de usar encapsulate deve-se usar apenas abstraction of.

Vamos agora atualizar o exemplo, descrito na seção que abordou o problema da primeira versão da biblioteca, para usar a solução proposta.

```
Class Comando
    syntax:
        Com
    semantics:
        executar _ : Com → Action
End Class
```

A classe Comando é exatamente a mesma especificada anteriormente, pois como vimos na seção sobre a Semântica de Ações Orientada a Objetos, ela é apenas a classe abstrata que servirá de base para as classes especializadas.

```
Class Selecao
    extending Comando
    using C1:Comando, C2:Comando, E:Expressao,
    objSel:LFL.Semantics.Command.Shared.Selection
    syntax:
        Com ::= "if" E "then" C1 "else" C2 "end-if"
    semantics:
        executar ⟦ "if" E "then" C1 "else" C2 "end-if" ⟧ =
            objSel.execute-if-then-else(encapsulate avaliar E,
            encapsulate executar C1, encapsulate executar C2)
End Class
```

Agora, ao instanciarmos o objeto *objSel*, não precisamos mais passar como parâmetros as classes Expressao e Comando. Isso porque ao usarmos o método execute-if-then-else, passamos as *abstrações* criadas usando a chamada encapsulate e os métodos criados na própria linguagem. Dessa forma, os métodos da linguagem são usados apenas na especificação dela, enquanto na biblioteca as *abstrações* são executadas sem dependência alguma dos métodos criados pelo usuário.

O grande ponto está em criar *abstrações* usando os próprios métodos definidos na especificação da linguagem. Isso significa que os métodos do usuário não são mais passados para a biblioteca e na verdade são traduzidos para a própria notação de *ações*, provida pelo formalismo orientado a objetos. Como essas *ações* estão na forma de *abstrações*, a partir do uso da ação enact, disponibilizada pela faceta reflexiva, é possível interpretá-las e executá-las.

Com essas mudanças acreditamos que além de melhorar a leitura e escrita, usando a LFL, a implementação da biblioteca em formalismos como a Semântica de Ações baseada em Módulos [DM01] e a baseada em Componentes [MM01] torna-se mais viável.

4.3 Novas Classes da LFL

A seguir apresentaremos as classes da LFL versão 2. Nesta versão disponibilizamos classes que implementam características do paradigma imperativo e classes que consideramos de uso comum em vários paradigmas de linguagens de programação. Usaremos figuras para representar cada nodo da LFL. Nestas figuras, as classes implementadas estão representadas por elipses acinzentadas, enquanto os demais elementos representão os níveis acima das classes implementadas.

4.3.1 Novas Classes do Nodo *Declaration*

O nodo *Declaration* é responsável por implementar classes que expressam a semântica de declarações. Veremos a implementação de classes que podem ser usadas na declaração de constantes, variáveis, variáveis com atribuição e seqüência de declarações. A estrutura hierárquica dessas classes pode ser observada na Figura 4.3.

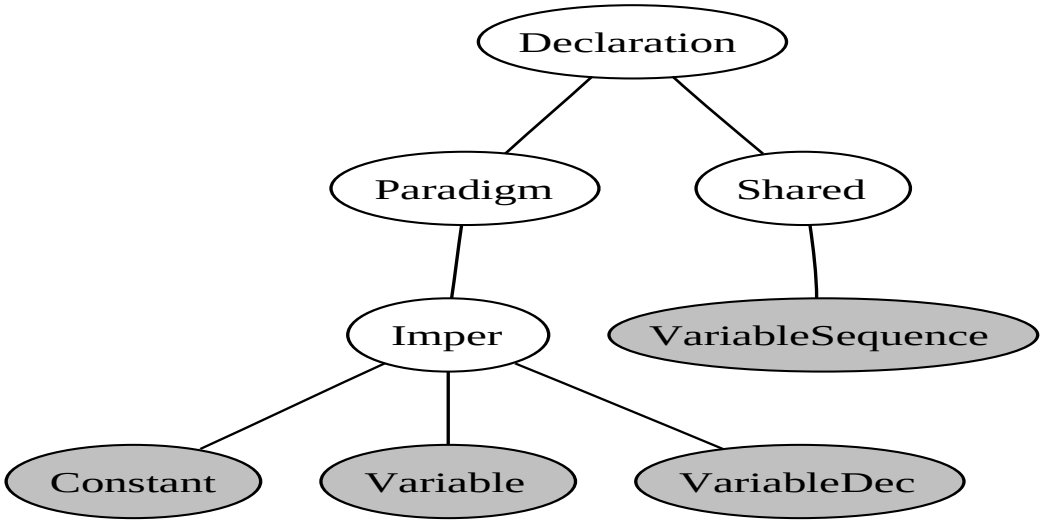

Figura 4.3: Classes implementadas no nodo *Declaration*

4.3.1.1 Classe *Constant*

A classe Constant representa as declarações de constantes comunmente encontradas em linguagens de programação imperativas, por isso ela está abaixo no nodo *Imper*. O método elaborate-constant recebe dois parâmetros, são eles: os objetos I:Token e Y:Yielder. O primeiro é referente ao nome da variável que será usada na declaração da constante, e o segundo é uma expressão abstraída, que quando computada gera o dado transitório que será associado à constante.

```
Class Constant
    locating LFL.Declaration.Paradigm.Imper.Constant
    using I:Token, Y:Yielder
    semantics:
        elaborate-constant(I, Y) =
            | enact Y
            then
            | bind token I to the value
End Class
```

4.3.1.2 Classe *Variable*

A classe Variable implementa o método elaborate-variable, o qual representa uma declaração de variável com atribuição. Essa classe também pertence ao nodo *Imper*, por ser mais uma característica evidente do paradigma imperativo. O método desta classe usa três parâmetros, são eles: os objetos I:Token, T:Sort e Y:Yielder. O *yielder* representa uma expressão abstraída, a qual após executada gera o dado que será armazenado na posição de memória alocada para a variável. Novamente o nome da variável é representado por um *token*, que por sua vez é atrelada a uma posição de memória alocada de acordo com o tipo de dado representado pelo *sort* em questão.

```
Class Variable
    locating LFL.Declaration.Paradigm.Imper.Variable
    using I:Token, T:Sort, Y:Yielder
    semantics:
        elaborate-variable(I, T, Y) =
            enact Y and allocate a cell [ T ] then
            | bind I to the given cell
            and
            | store the given value in the given cell
End Class
```

4.3.1.3 Classe *VariableDec*

Assim como na classe anterior, a classe VariableDec também representa uma declaração de variável, mas nesse caso sem atribuição. A variável representada pelo *token* I é atrelada à posição de memória alocada para o *sort* de dado T. Isto é implementado no método elaborate-variable e esta é mais uma classe de declaração do nodo *Imper*.

```
Class VariableDec
    locating LFL.Declaration.Paradigm.Imper.VariableDec
    using I:Token, T:Sort
    semantics:
        elaborate-variable(I, T) =
            | allocate a cell [ T ]
            then
            | bind token I to the given cell
End Class
```

4.3.1.4 Classe *VariableSequence*

A classe VariableSequence representa a seqüência de declarações de variáveis e como isso é possível de ocorrer em mais de um paradigma, ela foi implementada no nodo *Shared* das declarações. O método elaborate-var-sequence recebe dois *yielders* como parâmetros. Ambos são objetos que carregam declarações abstraídas e após computados geram os *bindings* das declarações executadas.

```
Class VariableSequence
    locating LFL.Declaration.Shared.VariableSequence
    using Y1:Yielder, Y2:Yielder
    semantics:
        elaborate-var-sequence(Y1,Y2) =
            | enact Y1 before enact Y2
End Class
```

4.3.2 Novas Classes do Nodo *Command*

Os comandos disponíveis na LFL são implementados na classe *Command*. Na biblioteca são disponibilizados os seguintes comandos: atribuição, laços de repetição, seleção e seqüência de comandos. Tais comandos, bem como suas posições na estrutura hierárquica, podem ser visualizados na Figura 4.4.

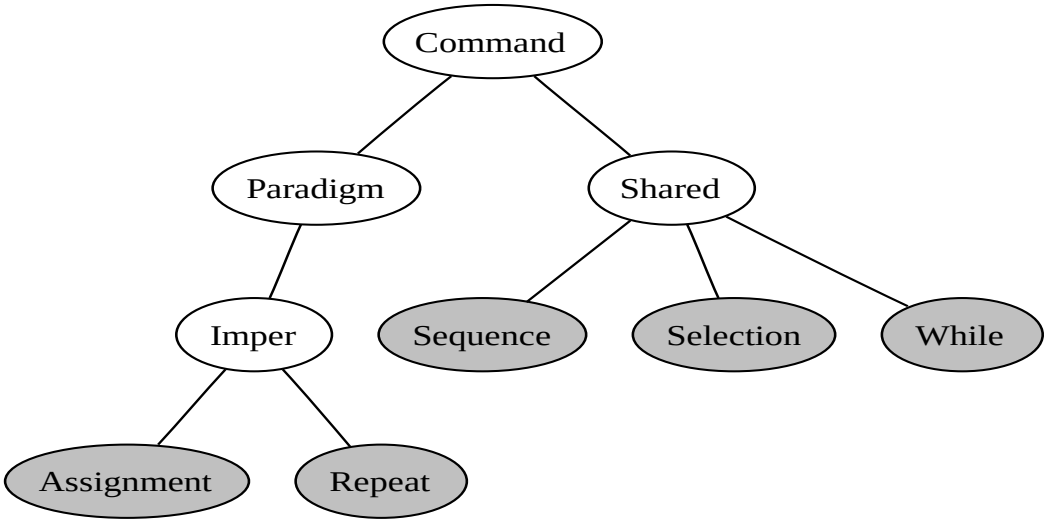

Figura 4.4: Classes implementadas no nodo *Command*

4.3.2.1 Classe *Assignment*

O comando de atribuição é representado na classe Assignment pelo método execute-assignment, o qual recebe como parâmetros um *token* e um *yielder*. O *yielder* carrega uma expressão abstraída e quando executada gera o valor que será armazenado na posição de memória atrelada à variável representada pelo *token* I. Como a atribuição é um comando característico de linguagens imperativas, esta classe é armazenada no nodo *Imper*.

```
Class Assignment
    locating LFL.Command.Paradigm.Imper.Assignment
    using I:Token, Y:Yielder
    semantics:
        execute-assignment(I, Y) =
            | enact Y
            then
            | store the value in the cell bound to token I
End Class
```

4.3.2.2 Classe *Repeat*

O laço de repetição estilo repeat do Pascal ou do-while do C é expresso na classe Repeat, a qual pertence ao nodo *Imper*. O comando representado nesta classe tem por característica executar um comando, em seguida verifica se a expressão é verdadeira e o comando é novamente executando enquanto a expressão satisfaz a condição por ela imposta. Este

comportamento é descrito no método execute-repeat, onde o objeto Y_1:Yielder representa o comando abstraído e o objeto Y_2:Yielder a expressão abstraída.

```
Class Repeat
    locating LFL.Command.Paradigm.Imper.Repeat
    using Y1:Yielder, Y2:Yielder
    semantics:
        execute-repeat(Y1,Y2) =
            unfolding
            | enact Y1 and then enact Y2
            | | then unfold else complete
End Class
```

4.3.2.3 Classe *Sequence*

A seqüência de comandos é expressa pela classe Sequence, a qual implementa o método execute-sequence, que por sua vez executa o primeiro comando e quando este termina em seguida executa o segundo. As abstrações dos comandos são representadas pelos objetos Y_1 e Y_2. Esta classe pertence ao nodo *Shared* por implementar uma característica comum a vários paradigmas.

```
Class Sequence
    locating LFL.Command.Shared.Sequence
    using Y1:Yielder, Y2:Yielder
    semantics:
        execute-sequence(Y1,Y2) =
            | enact Y1 and then enact Y2
End Class
```

4.3.2.4 Classe *Selection*

A classe Selection implementa comandos de seleção do tipo: if-then e if-then-else, os quais são implementados respectivamente nos métodos execute-if-then e execute-if-then-else. Tais estruturas de seleção estão presentes em inumeras linguagens, por isso esta classe encontra-se no nodo *Shared*. O método execute-if-then-else pode receber uma expressão abstraída e dois comandos abstraídos, como também pode receber apenas expressões abstraídas. Isto é possível porque usamos objetos que encapsulam abstrações de ações e em ambos os casos a primeira abstração é computada e caso resulte em valor verdadeiro a segunda

abstração é executada, caso contrário é executada a terceira. O mesmo comportamento é observado no método execute-if-then, mas nesse caso temos apenas duas abstrações, onde a primeira representa a condição e a segunda um comando ou expressão.

```
Class Selection
    locating LFL.Command.Shared.Selection
    using Y1:Yielder, Y2:Yielder, Y3:Yielder
    semantics:
        execute-if-then(Y1,Y2) =
            | enact Y1 then enact Y2 else complete
        execute-if-then-else(Y1,Y2,Y3) =
            | enact A1 then enact Y2 else enact Y3
End Class
```

4.3.2.5 Classe *While*

Um laço de repetição é expresso no método execute-while da classe While, a qual está localizada no nodo *Shared*. Os objetos Y_1 e Y_2 representam respectivamente uma expressão abstraída e um comando abstraído. Enquanto a expressão é verdadeira o comando é executado.

```
Class While
    locating LFL.Command.Shared.While
    using Y1:Yielder, Y2:Yielder
    semantics:
        execute-while(Y1,Y2) =
            unfolding
            | enact Y1 then
            | | enact Y2 and then unfold else complete
End Class
```

4.3.3 Novas Classes do Nodo *Expression*

Avaliações de expressões aritméticas, expressões lógicas e identificadores, bem como operadores lógicos e valores verdade são implementados nas classes que compõem o nodo *Expression*, o qual tem sua estrutura hierárquica representada na Figura 4.5. Neste nodo foram implementadas apenas características comuns a diversos paradigmas, por isso veremos apenas classes pertencentes ao nodo *Shared*. Nas classes do nodo *Expression* é

comum usarmos operações definidas e disponibilizadas em [Mos92] para dar o resultado da avaliação de expressões.

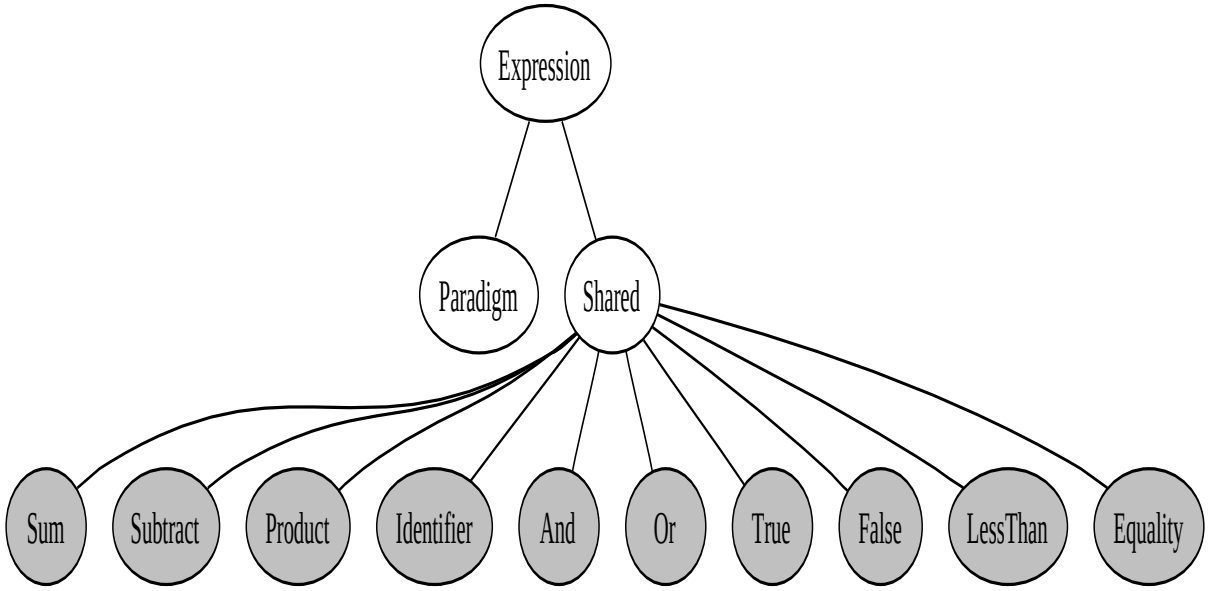

Figura 4.5: Classes implementadas no nodo *Expression*

4.3.3.1 Classe *Sum*

A classe Sum implementa a soma de duas expressões. As abstrações destas expressões aritméticas são representadas pelos objetos Y_1 e Y_2. Após a execução das expressões abstraídas, no método evaluate-sum, a soma delas é efetuada usando a operação sum, a qual é disponibilizada pela Semântica de Ações [Mos92].

```
Class Sum
    locating LFL.Expression.Shared.Sum
    using Y1:Yielder, Y2:Yielder
    semantics:
        evaluate-sum(Y1,Y2) =
            | enact Y1 and enact Y2
            then
            | give sum(the given integer #1, the given integer #2)

End Class
```

4.3.3.2 Classe *Subtract*

De forma semelhante a classe Sum, a classe Subtract implementa a diferença entre duas expressões aritméticas no método evaluate-sub. Após a avaliação dos objetos Y_1 e Y_2, a diferença entre as duas expressões é dada pela aplicação da operação difference.

```
Class Subtract
    locating LFL.Expression.Shared.Subtract
    using Y1:Yielder, Y2:Yielder
    semantics:
        evaluate-sub(Y1,Y2) =
            | enact Y1 and enact Y2
            then
            | give difference(the given integer #1, the given integer #2)
End Class
```

4.3.3.3 Classe *Product*

A classe Product efetua o produto de duas expressões. Isto é implementado no método evaluate-prod da classe em questão e após a avaliação das abstrações de expressões aritméticas contidas nos objetos Y_1 e Y_2, o produto é conseguido pela aplicação da operação product entre o resultado das duas expressões.

```
Class Product
    locating LFL.Expression.Shared.Product
    using Y1:Yielder, Y2:Yielder
    semantics:
        evaluate-prod(Y1,Y2) =
            | enact Y1 and enact Y2
            then
            | give product(the given integer #1, the given integer #2)
End Class
```

4.3.3.4 Classe *Identifier*

No método evaluate-identifier da classe Identifier é implementada a avaliação de uma variável. A variável a ser avaliada é representada pelo objeto I, o qual é um *token*. Como resultado é obtido o valor atrelado ao identificador I, ou o valor armazenado na célula de memória atrelada ao identificador I.

```
Class Identifier
    locating LFL.Expression.Shared.Identifier
    using I:Token
    semantics:
        evaluate-identifier(I) =
            | give the value bound to I
            or
            | give the value stored in the cell bound to I
End Class
```

4.3.3.5 Classe *And*

A classe And implementa a operador lógico and. Isto é feito no método evaluate-and, o qual recebe dois objetos como parâmetros. Os objetos Y_1 e Y_2 são expressões lógicas abstraídas. A primeira é avaliada, se for verdadeira então o resultado final será o resultado da avaliação da segunda, caso contrário o resultado final é falso.

```
Class And
    locating LFL.Expression.Shared.And
    using Y1:Yielder, Y2:Yielder
    semantics:
        evaluate-and(Y1,Y2) =
            enact Y1 then
            | check (the given value) and then enact Y2
            or
            | check not (the given value) and then give false
End Class
```

4.3.3.6 Classe *Or*

O operador lógico or é implementada no método evaluate-or da classe Or. De maneira similar à classe And, o método da classe Or também processa dois objetos que carregam abstrações de expressões lógicas. Se a avaliação de Y_1 resulta em valor verdadeiro, então esse é o resultado final da expressão, caso contrário o resultado final será o resultado da avaliação de Y_2.

```
Class Or
    locating LFL.Expression.Shared.Or
    using Y1:Yielder, Y2:Yielder
    semantics:
        evaluate-or(Y1,Y2) =
            enact Y1 then
            | check (the given value) and then give true
            or
            | check not (the given value) and then enact Y2
End Class
```

4.3.3.7 Classe *True*

A classe True implementa o valor verdade true no método evaluate-true. O valor true é gerado como um dado transitório para as demais ações. Nesta classe nenhum objeto é usado.

```
Class True
    locating LFL.Expression.Shared.True
    semantics:
        evaluate-true() =
            give true
End Class
```

4.3.3.8 Classe *False*

A classe False implementa o valor verdade false no método evaluate-false. De forma semelhante a classe anterior, o valor false também é gerado como um dado transitório e esta classe também não instancia nenhum objeto.

```
Class False
    locating LFL.Expression.Shared.False
    semantics:
        evaluate-false() =
            give false
End Class
```

4.3.3.9 Classe *LessThan*

No método evaluate-less-than da classe LessThan é implementada a operação lógica que verifica se um valor é menor que outro. Os objetos Y_1 e Y_2 representam expressões

aritméticas abstraídas e após a avaliação dessas expressões o resultado final é obtido pela aplicação da operação less, disponibilizada em [Mos92].

```
Class LessThan
    locating LFL.Expression.Shared.LessThan
    using Y1:Yielder, Y2:Yielder
    semantics:
        evaluate-less-than(Y1,Y2) =
            | enact Y1 and enact Y2
            then
            | give less(the given integer #1, the given integer #2)
End Class
```

4.3.3.10 Classe *Equality*

A classe Equality implementa a operação lógica de igualdade entre expressões no método evaluate-equality. As abstrações de expressões aritméticas contidas nos objetos Y_1 e Y_2 são avaliadas e o resultado se há igualdade entre as expressões é dado pela operação equal.

```
Class Equality
    locating LFL.Expression.Shared.Equality
    using Y1:Yielder, Y2:Yielder
    semantics:
        evaluate-equality(Y1,Y2) =
            | enact Y1 and enact Y2
            then
            | give equal(the given integer #1, the given integer #2)
End Class
```

4.4 Considerações finais

Apresentamos uma nova versão da LFL, a qual chamamos de LFL versão 2. Na versão anterior, a biblioteca usava passagem de parâmetros entre classes que por muitas vezes tornava a especificação mais difícil de ler. A nova versão da biblioteca resolveu o problema trocando a passagem de parâmetros pelo uso de abstrações de ações e da faceta reflexiva.

Antes, os métodos especificados na linguagem do usuário deviam ser passados como parâmetros. Para que os métodos disponibilizados pela LFL fossem capazes de definir a semântica desejada, de acordo com as classes e métodos especifcados pelo usuário.

Agora, as ações geradas pelos métodos especificados pelo usuário são transformadas em *abstrações*. Dessa forma, as ações podem ser computadas para definir a semântica desejada sem usar a antiga passagem de parâmetros entre classes, porém usando parâmetros conhecidos pela Notação de Ações da Semântica de Ações Orientada a Objetos.

Assim como na abordagem construtiva, fica a critério do usuário da biblioteca quais classes usar na especificação de uma determinada linguagem. As classes apropriadas devem ser escolhidas do repositório da LFL de acordo com as características semânticas desejadas e também levando em consideração as classes disponibilizadas pela biblioteca.

Além disso, o uso da biblioteca é significativamente similar ao uso da abordagem construtiva. A estrutura da LFL, separando conceitos por paradigmas de programação e conceitos comuns é bastante parecido com a divisão dos construtores, proposta pela abordagem construtiva. Por fim, consideramos viável o uso da LFL como uma abordagem para a definição de linguagens independente de sintaxe, pois o fato dela separar definições semânticas de definições sintáticas e disponibilizar uma hierarquia de classes previamente definidas tornou isso possível.

CAPÍTULO 5

MAUDE OBJECT-ORIENTED ACTION TOOL

Neste capítulo apresentaremos MOOAT (*Maude Object-Oriented Action Tool*), uma ferramenta que proporciona um ambiente executável para a especificação formal de linguagens usando a Semântica de Ações Orientada a Objetos. MOOAT foi desenvolvida usando a MMT, conforme a MSOS da Notação de Ações descrita em [Mos99] e seguindo as idéias propostas por [Car02].

O desenvolvimento de MOOAT foi inspirado por MAT (*Maude Action Tool*) [dOB01, dOBHMM00] e na sua implementação que usa um interpretador MSOS [Mos04], o qual foi desenvolvido usando Lógica de Reescrita [MOM93] em Maude [CDE+05] versão 1.

Além de prover um ambiente executável para especificações usando a Semântica de Ações Orientada a Objetos, MOOAT também tem por objetivo testar e provar as novas características da LFLv2. É importante ressaltar que a ferramenta possibilita a especificação de linguagens independentemente de sintaxe, como veremos nos estudos de caso apresentados nas seções 6.2 e 6.3.

5.1 Notação

Uma especificação em Semântica de Ações Orientada a Objetos é um conjunto finito de classes [Car02]. Tais classes criam uma hierarquia necessária para a especificação de uma determinada linguagem ou biblioteca. A relação entre essas classes é determinada de acordo com os objetos que são instanciados, bem como a posição onde elas se encontram na hierarquia especificada. Árvores sintáticas, na notação **BNF** usada na MMT, são usadas para definir a estrutura das frases da linguagem, e a semântica é dada pelo uso da Notação de Ações, de forma similar a descrita em [Mos99].

MOOAT é uma extensão conservativa de Full Maude [Dur99, DM99] e MMT [dR05, CB05, CB06]. Por isso, possui as mesmas limitações de Maude, como é descrito em

[CDE+05] e também as mesmas da MMT, como foi explicado na Seção 3.1.3 e detalhado em [dR05]. Por causa dessas limitações, a sintaxe abstrata de MOOAT possui algumas diferenças em relação à Semântica de Ações Orientada a Objetos, no entanto as duas são bastante similares. A sintaxe abstrata de MOOAT é mostrada a seguir.

5.1.1 Classes

Em (1) é definida a estrutura usada para a definição de uma classe. Onde a diretiva class indica o começo da classe e endclass o seu fim. O nome da classe é determinado por *ClassName*, o qual na verdade representa os identificadores aceitos por Maude e seguido da diretiva is é dado o corpo da classe.

O corpo da classe é constituído basicamente pela declaração das classes bases, regra (2), e pela definição da classe em si, regra (3). Repare que mais de uma classe base pode ser especificada, assim como as partes sintáticas e semânticas podem aparecer mais de uma vez na definição da classe. Tais partes serão detalhadas na próxima seção.

(1) *ClassModule* ::= "class" *ClassName* "is"
⟨ *ClassExtends* ⟩* ⟨ *ClassDefinition* ⟩* "endclass"

(2) *ClassExtends* ::= "extends" *ClassName* ⟨ "," *ClassName* ⟩* "."

(3) *ClassDefinition* ::= ⟨ *SyntacticPart* ⟩* ⟨ *SemanticPart* ⟩*

Se compararmos a notação da ferramenta com a notação do formalismo notaremos algumas diferenças, dentre as quais ressaltamos a mudança de extending para extends e a exclusão da diretiva using.

A primeira alteração deve-se ao fato de já existir uma diretiva extending em Maude e por causa do problema de pré-regularidade, descrito na seção 3.1.3, não é possível sobrecarregar o uso dessa diretiva, por isso foi necessário mudar para extends.

Já a exclusão da instanciação inicial de objetos deve-se ao fato de que ela foi transferida para a definição dos métodos, introduzindo o conceito de criação automática de objetos, nas próximas seções deixaremos mais clara essa transição.

A seguir temos um exemplo de definição de classe em MOOAT:

```
class ExpLang is
 extends Sum, Difference .
 extends Prod .
endclass
```

A classe `ExpLang` especifica uma linguagem hipotética de expressões aritméticas, onde as classes `Sum`, `Difference` e `Prod` são as classes que compõem a linguagem.

5.1.2 Definição das Classes

A definição de uma classe, regra (3), é composta de uma parte que define a sintaxe, regra (4), e outra que define a semântica, regra (5). A sintaxe segue a declaração sintática na forma BNF introduzida pela MMT. Já a declaração semântica pode ser composta por funções semânticas, as quais chamaremos de métodos abstratos, por equações semânticas, que aqui chamamos apenas de métodos, ou por ambos.

Em (6) especificamos que mais de um método abstrato pode ser definido. Tal método é na verdade a assinatura do método que será especificado nas subclasses ou na própria classe e pode resultar em uma ação ou em um *sort* de dado, como é mostrado em (7). Os *tokens* aceitos, (8), são os mesmos aceitos por Maude e como dito anteriormente devem ser usados para dar a assinatura de um método.

Na parte semântica podemos definir quantos métodos forem necessários, como é mostrado em (9). Um método, regra (10), implementa a semântica de um determinado conceito de uma linguagem de programação usando ações ou combinadores de ações. Tal método deve ser usado para especificar a semântica de uma sintaxe abstrata que foi definida na classe. Quando essa sintaxe abstrata é usada na declaração do método, os *sorts* sintáticos devem ser trocados por declarações de objetos.

Essas declarações devem seguir a sintaxe apresentada em (11), onde *Token* representa o nome do objeto e *SyntacticSort* representa o *sort* sintático que este objeto representará. Dessa forma introduzimos o conceito de criação automática dos objetos usados nos métodos e por isso a diretiva using não foi usada nesta implementação. Tais objetos são usados para computar as ações que expressam a semântica desejada.

(4) *SyntacticPart* ::= *SyntacticSort* “.” | *SyntacticSort* “::=” *syntax-tree* “.”

(5) *SemanticPart* ::= ⟨ *SemanticFunctions*⟩* ⟨ *SemanticEquations*⟩*

(6) *SemanticFunctions* ::= ⟨ *SemanticFunction*⟩+

(7) *SemanticFunction* ::=
“absmethod” *FunctionName Tokens* “->” “Action” “.” |
“absmethod” *FunctionName Tokens* “->” *Data* “.”

(8) *Tokens* ::= *TokenName* ⟨ “,” *Tokens*⟩*

(9) *SemanticEquations* ::= ⟨ *SemanticEquation*⟩+

(10) *SemanticEquation* ::=
“method” *FunctionName syntax-tree-with-objects* “=” *Action* “.”
“method” *FunctionName syntax-tree-with-objects* “=” *Data* “.”

(11) *ObjectDeclaration* ::= *Identifier* “:” *SyntacticSort*

A seguir veremos um exemplo de definição de declaração de constantes, onde a notação está mais clara:

```
class Declaration is
 Dec .
 absmethod elaborate Dec -> Action .
endclass

class ConstantDeclaration is
 extends Declaration .
 Dec ::= const Token = Exp .
 method elaborate (const I:Token = E:Exp) =
    evaluate E:Exp then
    bind I:Token to given value .
endclass
```

Neste exemplo definimos duas classes: `Declaration` e `ConstantDeclaration`. A primeira corresponde à classe base, onde o *token* `Dec` e o método abstrato `elaborate` são definidos. O método definido cria uma assinatura do *sort* sintático `Dec` para uma ação `Action`.

A segunda classe é a classe especializada, ou seja, a classe que implementa uma determinada característica da linguagem especificada. Nesse caso, `ConstantDeclaration` é uma extensão de `Declaration` e é usada para definir uma declaração de constante. O *sort* `Dec` é sobrecarregado com a declaração de constante usada na linguagem, e o método

`elaborate` também é sobrecarregado para expressar a semântica correta da declaração representada.

Repare que os *sorts* sintáticos `Token` e `Exp` foram usados. No caso do primeiro, veremos que ele é definido pela notação de ações da ferramenta e como trata-se de um Data pode ser usado tranqüilamente nos métodos. Já no caso do segundo assumimos que o usuário definiu em alguma outra classe o *sort* `Exp`. Esses mesmos elementos sintáticos devem ser usados na declaração dos objetos, pois a ferramenta identifica o tipo do objeto a ser criado pelo *sort* sintático especificado. A notação de declaração de objetos introduzida pela ferramenta deve ser usada tanto na parte esquerda do método como na parte direita. Na esquerda temos a implementação do método para uma determinada árvore sintática e na direita temos a ação correspondente.

Na Semântica de Ações, a ação correspondente a evaluate E é determinada por equações semânticas sobre a sintaxe representada por E. No caso da Semântica de Ações Orientada a Objetos, evaluate é um método que pode ser aplicado a um objeto E, estruturado exatamente como uma equação semântica e que produz a mesma ação. Nesse trabalho criamos um ambiente para representar e testar essas especificações em Semântica de Ações Orientada a Objetos, para que isso fosse possível implementamos a Notação de Classes mostrada até aqui e também implementamos a Notação de Ações descrita em [Mos99], conforme a sintaxe mostrada a seguir.

5.1.3 Notação de Ações

As regras de (12) a (16) definem a sintaxe da Notação de Ações a ser usada no corpo dos métodos em MOOAT. O conjunto das ações é expresso em (12). Os *yielders* e os dados usados são definidos em (13) e (14), respectivamente.

Em (15) temos os dados implementados, os quais precisaram usar uma função de coerção (< >) devido ao problema de *ad-hoc overloading* citado na seção 3.1.3. Os *sorts* correspondentes aos dados implementados são descritos em (16). Nas próximas seções veremos como a Notação de Ações foi implementada usando a MMT para expressar a semântica da Notação de Ações em Semântica Operacional Estrutural Modular e como a

Notação de Classes foi especificada em Maude.

(12) *Action* ::= *Action* "or" *Action* | "fail" | "commit" | *Action* "and" *Action* |
"complete" | "indivisibly" *Action* | *Action* "and then" *Action* |
Action "trap" *Action* | "escape" | "unfolding" *Action* | "unfold" |
"diverge" | "give" Yielder" | "regive" | "choose" *Yielder* |
"check" *Yielder* | *Action* "then" *Action* | "escape with" *Yielder* |
"bind" *Yielder* "to" *Yielder* | "rebind" | "unbind" *Yielder* |
"produce" *Yielder* | "furthermore" *Action* | *Action* "moreover" *Action* |
Action "hence" *Action* | *Action* "before" *Action* |
"store" *Yielder* "in" *Yielder* | "unstore" *Yielder* | "reserve" *Yielder* |
"unreserve" *Yielder* | "enact" *Yielder* |
"indirectly bind" *Yielder* "to" *Yielder* | "indirectly produce" *Yielder* |
"redirect" *Yielder* "to" *Yielder* | "undirect" *Yielder* |
"recursively bind" *Yielder* "to" *Yielder* |
Action "else" *Action* | "allocate" *Yielder*

(13) *Yielder* ::= *Data* | "a" *Yielder* | "the" *Yielder* | "nothing" |
"the" *DataSort* "yielded by" *Yielder* | "it" | "them" |
"given" *DataSort* | "given" *DataSort* # *Int* |
"current bindings" | "the" *DataSort* "bound to" *Yielder* |
Yielder "receiving" *Yielder* |
"current storage" | "the" *DataSort* "stored in" *Yielder* |
"application" *Yielder* "to" *Yielder* | "closure" *Yielder* |
"encapsulate" *Action* | "indirect closure" *Yielder*

(14) *Data* ::= *Datum* | *DataSort*

(15) *Datum* ::= "none" | "unknown" | "uninitialized" | *Abstraction* |
"<" *Int* ">" | "<" *Boolean* ">" | "<" *Token* ">" | "<" *Cell* ">" |
"<" *Transients* ">" | "<" *Bindings* ">" | "<" *Storage* ">" |

(16) *DataSort* ::= "integer" | "truth-value" | "token" | "cell" |
"abstraction" | "value"

5.1.4 Exemplo

Nesta seção mostraremos um exemplo simples de uso de MOOAT, onde especificaremos uma linguagem de uma simples calculadora aritmética. As operações especificadas são: soma, subtração, multiplicação e divisão. A estrutura hierárquica da especificação pode ser conferida na Figura 5.1:

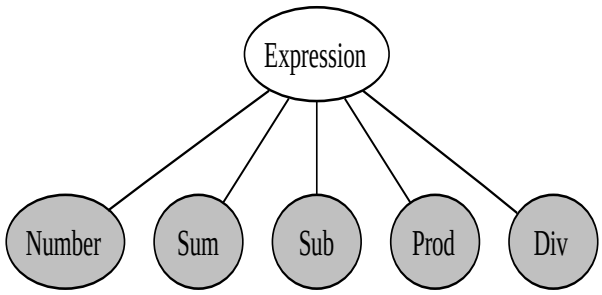

Figura 5.1: Estrutura hierárquica da Simples Calculadora

```
(class Expression is
 Exp .
 absmethod evaluate Exp -> Action .
endclass)
```

Começamos a especificação pela classe abstrata `Expression`. Nela especificamos o *sort* `Exp` e o método abstrato `evaluate`, os quais serão sobrecarregados pelas subclasses dela. Um método abstrato é definido ao usarmos a diretiva `absmethod`.

```
(class Number is
 extends Expression .
 Exp ::= Int .
 method evaluate (I:Int) =
    give < I:Int > .
endclass)
```

A classe `Number` é uma subclasse de `Expression`, como é definido pela diretiva `extends`. A árvore sintática `Exp ::= Int .` define que um número inteiro é um terminal de `Exp`. O objeto `I:Int`, de *sort* `Int`, é usado para especificar a semântica da avaliação de um número inteiro, onde ele é apenas retornado pela ação `give`. Repare que o método `evaluate` agora é sobrecarregado para expressar a Semântica de Ações do comportamento representado, isso é possível pelo uso da diretiva `method` e do uso dos objetos necessários para o processamento da sintaxe implementada pelo método especificado.

```
(class Sum is
 extends Expression .
 Exp ::= Exp '+ Exp .
 method evaluate (E1:Exp '+ E2:Exp) =
    (evaluate E1:Exp and evaluate E2:Exp)
```

```
    then give sum (given integer # 1, given integer # 2) .
endclass)

(class Sub is
 extends Expression .
 Exp ::= Exp '- Exp .
 method evaluate (E1:Exp '- E2:Exp) =
    (evaluate E1:Exp and evaluate E2:Exp)
    then give difference (given integer # 1, given integer # 2) .
endclass)

(class Prod is
 extends Expression .
 Exp ::= Exp '* Exp .
 method evaluate (E1:Exp '* E2:Exp) =
    (evaluate E1:Exp and evaluate E2:Exp)
    then give product (given integer # 1, given integer # 2) .
endclass)

(class Div is
 extends Expression .
 Exp ::= Exp '/ Exp .
 method evaluate (E1:Exp '/ E2:Exp) =
    (evaluate E1:Exp and evaluate E2:Exp)
    then give quotiente (given integer # 1, given integer # 2) .
endclass)
```

As classes `Sum`, `Sub`, `Prod` e `Div` são similares, pois além de serem classes especializadas de `Expression`, cada uma delas implementa uma das operações suportadas pela calculadora. Nas quatro classes são especificadas as árvores sintáticas das operações implementadas e as sua respectivas semânticas usando a diretiva `method`.

Também são usados os objetos `E1:Exp` e `E2:Exp`, de *sort* `Exp`. Em todas essas classes os dois objetos são avaliados, pelo uso do método `evaluate`, e uma operação é aplicada sobre os dois números inteiros resultantes dessa avaliação para que o valor final seja conseguido.

As operações aritméticas de soma, subtração, multiplicação e divisão são respectivamente implementadas na Notação de Ações de MOOAT pelas seguintes operações auxiliares: `sum(_,_)`, `difference(_,_)`, `product(_,_)` e `quotiente(_,_)`.

Terminaria aqui o exemplo da especificação de uma calculadora em Semântica de Ações Orientada a Objetos usando MOOAT se Maude fosse capaz de executar os métodos definidos em cada classe separada. No entanto, para ser possível executar uma expressão

aritmética será necessário especificar uma classe que crie o ambiente necessário para a execução, como é mostrado na classe a seguir.

```
(class Simple-Calc is
 extends Number .
 extends Sum, Sub .
 extends Prod, Div .

 absmethod exp-test -> Action .
 method exp-test = (evaluate (((1 '+ 2) '* 3) '/ (5 '- 4))) .
endclass)
```

A classe `Simple-Calc` cria o ambiente necessário para executarmos nossas operações. Para que isso fosse possível precisamos estender as classes de nível mais baixo na hierarquia da linguagem, as quais foram representadas por elipses acinzentadas na Figura 5.1, sendo assim Maude consegue visualizar os métodos implementados por cada classe.

Pode-se também definir um método abstrato que resulta em uma ação, para efeitos de teste e/ou demonstração da linguagem. Tal método abstrato pode ser sobrecarregado para implementar uma expressão suportada, como foi exemplificado.

Veremos nas próximas seções a implementação do comando `compute`. Este comando é usado para executar uma ação em MOOAT, pois inicializa o ambiente de forma correta para a execução de um programa na linguagem especificada. O comando `rewrite` deve ser usado em conjunto com `compute` para que a ação seja computada por Maude.

Repare a seguir que podemos usar o método `exp-test`, definido na classe `Simple-Calc`, ou então passar direto uma expressão válida para o processamento. Tal parâmetro é encarado como uma ação, a qual é computada para gerar o resultado final.

```
Maude> (rew compute (exp-test) .)
rewrites: 10563922 in 41820ms cpu (51597ms real) (252604 rewrites/second)
rewrite in Simple-Calc :
  compute(exp-test)
result Conf :
  < gave(< 9 >):::  'Action,{unfolding = fail,bindings
    =(void).Map{Token,Data}, commitment = false,redirections =(void).Map{
    Indirection,Data},storage =(void).Map{Cell,Data},
    data =(1 |-> < 9 > +++ 2 |-> < 1 >)}>
```

```
Maude> (rew compute (evaluate (1 '+ 1)) .)
rewrites: 10563811 in 41850ms cpu (49924ms real) (252420 rewrites/second)
rewrite in Simple-Calc :
  compute(evaluate(1 '+ 1))
result Conf :
  < gave(< 2 >):::: 'Action,{unfolding = fail,bindings
   =(void).Map{Token,Data},commitment = false,redirections =(void).Map{
    Indirection,Data},storage =(void).Map{Cell,Data},
    data =(1 |-> < 1 > +++ 2 |-> < 1 >)}>
```

5.2 Implementação

A implementação de MOOAT é uma extensão conservativa de Full-Maude [Dur99, DM99] e MMT [dR05, CB05, CB06]. Onde a Notação de Ações é dada em módulos MSDF, os quais implementam as transições MSOS necessárias para a implementação das ações disponíveis. Já a Notação de Classes foi implementada para que o estilo de classes, proposto pela Semântica de Ações Orientada a Objetos, fosse aceito e por isso foi necessário criar operações e equações para interpretar essas classes.

Essa implementação foi possível visto que a modularidade em especificações MSOS está nos rótulos usados pelas transições. Por isso, foi possível implementar a Notação de Ações definida em [Mos99], usando os módulos MSDF da MMT. Em seguida implementamos a Notação de Classes, na qual é possível criar especificações no estilo da Semântica de Ações Orientada a Objetos. As ações definidas nas classes são especificadas usando a notação implementada por MOOAT, a qual contempla a criação automática de objetos. Devido a restrições de Maude, o ambiente que concentra os métodos definidos na especificação deve ser definido pelo usuário, no entanto `STATE` é que provê todas essas características mencionadas.

Veremos nessa seção como algumas transições da Notação de Ações foram implementadas, bem como a Notação de Dados correspondente. Também elucidaremos como as classes são tratadas pela ferramenta, ou seja, como alteramos o `LOOP-MODE` [CB05, CDE+05, DM99] para que essas classes fossem aceitas e transformadas em módulos de Maude, mesma técnica empregada nos módulos orientado a objetos de Full-Maude [DM99] e nos módulos MSDF da MMT [dR05]. Não esqueceremos de mencionar como a classe

`STATE` de [Car02] é tratada em MOOAT.

Para que fosse possível implementar algumas transições da Notação de Ações foi necessário adicionar duas novas operações para atuar sobre os mapas definidos pela MMT, são elas:

disjoint-union : (s, s')Map x (s,s')Map → Bool

Verifica se é possível efetuar a união disjunta entre dois mapas.

omitting_in_ : s x (s, s')Map → (s, s')Map

Omite o elemento s em (s, s')Map.

5.2.1 Notação de Dados

A Notação de Dados foi implementada usando um módulo MSDF chamado `DataNotation` e um módulo de sistema Maude chamado `DATANOTATION`. O primeiro foi usado para definir os *sorts* de dados usados pela Notação de Ações, já o segundo implementa a função `_:<_`, a qual é usada nas transições MSOS para verificar se um determinado dado pertence a um determinado *sort*. Por exemplo, `< 1 > :< value` retorna true, pois `value` é interpretado tanto como `integer` ou `truth-value` e `< cell(1) > :< abstraction` retornaria false, pois uma célula de memória não é uma abstração.

Como exemplos de *sorts* definidos pela Notação de Dados de MOOAT citamos: `Action`, `Yielder` e `Data`. Repare que esses *sorts* são os mesmos especificados na seção 5.1.3, a qual definiu a sintaxe da Notação de Ações de MOOAT.

Algumas operações auxiliares são definidas na Notação de Dados de MOOAT, para auxiliar a criação de especificações em Semântica de Ações Orientada a Objetos, são elas:

- `sum (Yielder, Yielder)` - soma dois números;
- `difference (Yielder, Yielder)` - subtrai dois números;
- `product (Yielder, Yielder)` - multiplica dois números;
- `quotiente (Yielder, Yielder)` - divide dois números;
- `less (Yielder, Yielder)` - verifica se um número é menor que outro;

- `greater (Yielder, Yielder)` - verifica se um número é maior que outro;
- `equal (Yielder, Yielder)` - verifica a igualdade entre dois números;
- `not Yielder` - efetua a negação de um valor verdade.

5.2.2 Notação de Ações

A sintaxe da Notação de Ações implementada por MOOAT foi especificada na seção 5.1.3, seguindo a notação descrita em [Mos99] para contemplar a notação proposta em [Car02]. Como vimos anteriormente, a modularidade da MSOS está na definição de transições rotuladas. Veremos a seguir que a implementação da Notação de Ações foi dividida em módulos MSDF para representar cada uma das facetas implementadas. Sendo assim, cada uma das facetas especifica os índices usados nos rótulos das transições.

Como as transições e os módulos implementados são na verdade traduções dos módulos e transições descritos em [Mos99] para a notação MSDF, implementada pela MMT, apenas introduziremos os módulos criados e algumas transições implementadas, para que fique claro como a Notação de Ações de MOOAT foi implementada usando a MMT. Não mostraremos todas as transições porque além de serem muitas, elas são bastante similares às descritas em [Mos99], sendo assim a descrição de algumas transições é suficiente para a compreensão da criação da ferramenta descrita por este trabalho.

5.2.2.1 Faceta Básica

A faceta básica é composta por seis módulos MSDF. No módulo `BasicData` são implementados os dados necessários para o processamento das transições desta faceta, como em Maude os valores true e false são implementados pelo *sort* Bool, fizemos uso dele.

A sintaxe das ações e dos *yielders* usados é especificada no módulo `BasicSyntax`. Já, `BasicOutcomes` foi usado para especificar o *sort* `Terminated` para os possíveis resultados de uma ação. Os quais nessa faceta são: `completed`, `escaped` ou `failed`. Em `BasicConfigurations` especificamos que uma ação pode ser um dos três resultados mencionados anteriormente, ou então pode ser uma composição de ações (`Action @ Action`),

veremos que esse combinador auxiliar é usado por `unfolding`.

O rótulo usado pela faceta básica é definido no módulo `BasicLabels` e é composto dos índices `commitment` e `unfolding`. O primeiro é um valor booleano usado para validar se uma ação efetuou um commit ou não e o segundo é usado para executar uma determinada ação por `unfold`, a qual foi armazenada ao usarmos o combinador `unfolding`. A definição do rótulo é mostrada a seguir:

```
Label = {commitment : Bool, commitment' : Bool,
         unfolding : Action, unfolding' : Action, ...} .
```

As transições dessa faceta foram implementadas no módulo `BasicTransitioins`. Dentre as quais mostraremos a implementação dos combinadores or, unfolding, unfold e dos *yielders* implementados. Para que ficasse mais clara a analogia entre a implementação de MOOAT usando MMT e as transições descritas em [Mos99], antes das transições implementadas inserimos alguns comentários que fazem referência a sua correspondente no documento de Mosses.

```
--- 1
            Action1 -{...}-> Action'1
 -- ------------------------------------------------------
 Action1 or Action2 : Action -{...}-> Action'1 or Action2 .

--- 2
            Action2 -{...}-> Action'2
 -- ------------------------------------------------------
 Action1 or Action2 : Action -{...}-> Action1 or Action'2 .

--- 4
 completed or Action2 : Action --> completed .

--- 5
 Action1 or completed : Action --> completed .

--- 6
 escaped or Action2 : Action --> escaped .

--- 7
 Action1 or escaped : Action --> escaped .

--- 8
 failed or Action2 : Action --> Action2 .
```

```
--- 9
 Action1 or failed : Action --> Action1 .
```

Repare que as transições 1 e 2 representam a escolha não determinística entre as ações Action1 e Action2. Em 4 e 5, se uma ação completou com sucesso, então o resultado final é completed independentemente da avaliação da outra ação. O mesmo comportamento acontece no caso de uma ação escapar, como é mostrado em 6 e 7. No entanto, se uma ação falhar a outra é executada, como pode ser observado em 8 e 9.

```
--- 34
 unfolding Action0 : Action --> Action0 @ Action0 .

--- 35
 Terminated @ Action0 : Action --> Terminated .

--- 36
 Action0 @ Action0 : Action -{unfolding = Action,
    unfolding' = Action0,-}-> Action0 .

--- 37
 unfold : Action -{unfolding = Action0, unfolding' = Action0,-}->
    Action0 .
```

O combinador de ações auxiliar (@) é usado pela transição 34 e o seu critério de parada é definido em 35, ou seja, a execução é terminada quando uma ação chega em um dos valores definidos pelo *sort* Terminated. Se a ação não chegou em um estado final ela é armazenada no índice unfolding primalizado e é executada, como é observado em 36. Essa ação armazenada poderá ser executada novamente quando usamos o combinador unfold, assim como é especificado em 37. Laços de repetição podem ser especificados ao usarmos os combinadores unfolding e unfold.

```
--- 39 e 40
 the DataSort yielded by Yielder : Yielder -->
    if (Yielder :< DataSort) then Yielder else nothing fi .
```

As transições 39 e 40 foram implementadas em apenas uma, onde se um *yielder*, avaliado para um Data, pertence ao *sort* especificado em DataSort, então ele é retornado e caso contrário nothing é retornado.

5.2.2.2 Faceta Funcional

O módulo `FunctionalData` especifica os dados introduzidos pela faceta funcional, ou seja, especificamos que os dados transitórios são do *sort* `Transients`, o qual é um mapeamento de um número inteiro (`Int`) para um determinado dado (`Datum`). Os dados transitórios são representados por `Transients` no rótulo implementado no módulo `FunctionalLabels`, onde os índices `data` e `data` primo carregam os dados transitórios atuais. Veja a implementação do rótulo a seguir:

```
Label = {data : Transients, data' : Transients, ...} .
```

No módulo `FunctionalSyntax` é especificada a sintaxe dos combinadores de ações desta faceta. Já o módulo `FunctionalOutcomes` redefine o *sort* `Terminated` para que aceite os valores introduzidos pelo *sort* `Completed`, os quais são: `completed`, para determinar que uma ação completou e `gave (Data)`, para especificar a produção de um dado transitório.

As transições funcionais são implementadas no módulo `FunctionalTransitions`. A seguir mostraremos como foi a implementação dos combinadores give e then, além de elucidar como funcionam os *yielders* given DataSort e given DataSort # Int.

```
--- regra auxiliar para avaliar um yielder
        Yielder -{...}-> Yielder'
 -- ----------------------------------------
 give Yielder : Action -{...}-> give Yielder' .

--- 1
 give Data : Action --> gave (Data) .

--- 2
 give nothing : Action --> failed .
```

Antes das regras de avaliação do combinador `give` foi necessário implementar uma regra auxiliar para avaliar o *yielder* produtor do dado que será usado nesta ação. Se este *yielder* realmente produz um dado (`Data`), ele é passado para a ação `gave (Data)` para que o dado transitório correspondente seja gerado. Porém, caso `nothing` seja produzido a ação falha. Tais comportamentos são observados nas transições 1 e 2, respectivamente.

```
--- 18
            Action1 -{...}-> Action'1
 -- ------------------------------------------------------------
 Action1 then Action2 : Action -{...}-> Action'1 then Action2 .

--- 19
        Transients' := (1 |-> Data1) / Transients
 -- ------------------------------------------------
 gave (Data1) then Action2 : Action -{data = Transients,
    data' = Transients',-}-> Action2 .

--- 20
        Transients' := (1 |-> Data1) / Transients
 -- ------------------------------------------------------
 gave (Data1) then Completed2 : Action -{data = Transients,
    data' = Transients',-}-> Completed2 .

--- 23
 Completed1 then failed : Action --> failed .

--- 24
 failed then Action2 : Action --> failed .
```

A implementação do combinador `then` começa na transição `18`. Nela, a primeira ação é avaliada e caso resulte em `gave (Data)` a segunda ação começa a ser avaliada, podendo usar o dado transitório produzido pela primeira ação, como pode ser observado em `19` e `20`. Se por ventura uma das duas ações falhar o resultado final também é falho, como é mostrado em `23` e `24`. A produção de mais de um dado transitório é efetuada em outras transições, as quais não foram apresentadas neste documento porque a explicação destas regras apresentadas é suficiente para o entendimento das demais analisando diretamente o código fonte.

```
--- 33
    Data := lookup (1, Transients)
 -- ------------------------------------------------------------------
 given DataSort : Yielder -{data = Transients, data' = Transients,-}->
    if (Data :< DataSort) then Data else given DataSort # 2 fi .

--- 34
    Data := lookup (Int, Transients)
 -- ------------------------------------------------
 given DataSort # Int : Yielder -{data = Transients,
    data' = Transients,-}-> if (Data :< DataSort)
```

```
then Data else given DataSort # (Int + 1) fi .
```

A transição `33` produz um dado de um determinado *sort*, representado por `DataSort`. Verificamos se esse dado é o primeiro elemento dos dados transitórios e caso não seja o controle é passado para a transição `34`, para que ele seja encontrado no mapa de dados transitórios atuais. Ambas as transições podem ser usadas para obter um dado de um determinado *sort*.

5.2.2.3 Faceta Declarativa

Assim como a faceta funcional, a faceta declarativa também é composta por cinco módulos MSDF. Os dados implementados por esta faceta estão no módulo `DeclarativeData`, onde o conjunto dos *bindings* é definido pelo *sort* `Bindings`, o qual é um mapeamento de um *token* (`Token`) para um *bindable* (`Data`).

Representamos os *bindables* usando o *sort* `Data`, pois caso definíssemos um possível *sort* `Bindable` cairíamos no problema de pré-regularidade, já que alguns dos *subsorts* seriam os mesmos para ambos.

Ainda neste módulo definimos que um `Token` é um elemento do *sort* `Qid` (*quoted identifier*), implementado por Maude. Dessa forma, `'a`, `'abc`, `'a1` e outras construções que sejam desse tipo de identificador podem ser usadas como *tokens* em MOOAT.

A sintaxe usada na implementação das transições desta faceta foi especificada no módulo `DeclarativeSyntax`. O módulo `DeclarativeOutcomes` redefine o *sort* `Completed`, introduzido no módulo `FunctionalOutcomes`, para que aceite a seguinte construção: `produced (Bindings)`, referente aos *bindings* produzidos por uma determinada ação.

O rótulo usado na faceta das declarações é definido no módulo `DeclarativeLabels`, o qual declara o índice `bindings` e o seu respectivo índice primalizado. Ambos os índices são do *sort* `Bindings` para representar o mapeamento dos *bindings* usados nos combinadores de ações implementados no módulo `DeclarativeTransitions`. Confira a seguir a definição do rótulo com os seus índices:

```
Label = {bindings : Bindings, bindings' : Bindings, ...} .
```

A seguir veremos detalhadamente a implementação da ação bind Yielder to Yielder, do combinador de ações hence e do *yielder* the DataSort bound to Yielder.

```
--- regras auxiliares para avaliar os yielders
        Yielder1 -{...}-> Yielder'1
 -- ---------------------------------------
 bind Yielder1 to Yielder2 : Action -{...}->
    bind Yielder'1 to Yielder2 .

        Yielder2 -{...}-> Yielder'2
 -- ---------------------------------------
 bind Yielder'1 to Yielder2 : Action -{...}->
    bind Yielder'1 to Yielder'2 .

--- 1
        Bindings' := (Token |-> Data) / Bindings
 -- ------------------------------------------------
 bind < Token > to Data : Action -{bindings = Bindings,
    bindings' = Bindings',-}-> produced (Token |-> Data) .

--- 2 (foi dividida em duas)
 bind nothing to Yielder : Action --> failed .

 bind Yielder to nothing : Action --> failed .
```

Primeiramente os dois *yielders* devem ser avaliados e caso um deles produza `nothing` a ação falha, como é mostrado nas duas regras que implementam a transição 2. Quando o `Yielder1` produz um *token* (`< Token >`) e o `Yielder2` um dado (`Data`), como na transição 1, o primeiro é amarrado ao segundo e um novo ambiente, contendo esse novo *bindings*, é produzido. Repare que a função de coerção (`< >`) foi necessária no uso da regra e não foi no mapeamento. Justamente porque se não fosse usada em ambos os casos produziria o problema de *ad-hoc overloading*.

```
--- 23
                Action1 -{...}-> Action'1
 -- ------------------------------------------------------------
 Action1 hence Action2 : Action -{...}-> Action'1 hence Action2 .

--- 24
        Bindings' := (Bindings1 / Bindings)
 -- ------------------------------------------------------------
 produced (Bindings1) hence Action2 : Action -{bindings = Bindings,
    bindings' = Bindings',-}-> Action2 .
```

```
--- 25
 produced (Bindings) hence Terminated2 : Action --> Terminated2 .

--- 27
 failed hence Action2 : Action --> failed .
```

A característica do combinador de ações `hence` é a execução das ações `Action1` e `Action2` seqüencialmente, de uma forma que os *bindings* da primeira ação são propagados para a segunda ação para a produção da informação final. Esse comportamento pode ser observado nas transições 23 e 24, sendo que na segunda transição mencionada o elemento `Bindings`' é o resultado da união dos *bindings* da primeira com os atuais e é passado para a execução da segunda, como explicamos anteriormente. Se a segunda ação for do *sort* `Terminated` prevalece esse resultado, como observado em 25. Em 27 mostramos que se a primeira ação falhar o resultado final é falho sem mesmo precisar executar a próxima ação.

```
--- regra auxiliar para avaliar um yielder
            Yielder -{...}-> Yielder'
 -- ------------------------------------------
 the DataSort bound to Yielder : Yielder -{...}->
    the DataSort bound to Yielder' .

--- 36 e 37
                def lookup (Token, Bindings),
                Data := lookup(Token, Bindings)
 -- ------------------------------------------------------------
 the DataSort bound to < Token > : Yielder -{bindings = Bindings,
    bindings' = Bindings,-}-> if (Data :< DataSort)
    then Data else nothing fi .

--- 38
                def lookup (Token, Bindings) =/= true
 -- ------------------------------------------------------------
 the DataSort bound to < Token > : Yielder -{bindings = Bindings,
    bindings' = Bindings,-}-> nothing .

--- 39
 the DataSort bound to nothing : Yielder --> nothing .
```

Se após avaliação do `Yielder` for produzido um *token* e tal identificador está presente no mapeamento dos *bindings* atuais, o dado amarrado a `Token` é retornado caso pertença

ao *sort* de dado especificado por `DataSort`, caso contrário `nothing` é retornado. Tal comportamento pode ser observado na regra que implementa as transições 36 e 37. Nada é produzido quando um determinado identificador não existe no mapeamento ou quando um determinado *yielder* é avaliado para `nothing`, como é mostrado pelas transições 38 e 39.

5.2.2.4 Faceta Imperativa

Quatro módulos MSDF foram necessários para a implementação da faceta imperativa. O *sort* `Storage`, usado para definir o *storage* usado pelas ações, foi implementado como um mapeamento de uma posição de memória (`Cell`) para um dado (`Data`) no módulo `ImperativeData`.

Assim como na faceta declarativa, na faceta imperativa os *storables* são representados pelo *sort* `Data` por causa do problema de pré-regularidade ao usarmos os mesmos *subsorts* em *sorts* diferentes.

Uma posição de memória é representada pela construção `cell (Int)`, onde o número inteiro representa o endereço de memória. Uma nova posição de memória pode ser criada ao usarmos a operação `new-cell` passando como parâmetro o `Storage` atual.

A sintaxe dessa faceta é implementada no módulo `ImperativeSyntax`, enquanto o rótulo é dado no módulo `ImperativeLabels`. De forma similar a faceta declarativa, o rótulo dessa faceta também possui dois índices: `storage` e `storage'`, os quais carregam a memória usada e fazem com que ela possa ser alterada nas especificações de MOOAT. Como vimos anteriormente, a memória é representada por `Storage` e é usada no rótulo da seguinte maneira:

```
Label = {storage : Storage, storage' : Storage, ...} .
```

O módulo `ImperativeTransitions` implementa as transições da faceta imperativa. Elucidaremos a implementação da ação store Yielder in Yielder e do *yielder* the DataSort stored in Yielder.

```
--- regras auxiliares para avaliar os yielders
```

```
        Yielder1 -{...}-> Yielder'1
 -- ---------------------------------------
 store Yielder1 in Yielder2 : Action -{...}->
    store Yielder'1 in Yielder2 .

        Yielder2 -{...}-> Yielder'2
 -- ---------------------------------------
 store Yielder'1 in Yielder2 : Action -{...}->
    store Yielder'1 in Yielder'2 .

--- 1
            def lookup (Cell, Storage),
            Storage' := (Cell |-> Data) / Storage
 -- ------------------------------------------------------------
 store Data in < Cell > : Action -{storage = Storage,
    storage' = Storage', commitment = Bool, commitment' = true,-}->
    completed .

--- 2
        def lookup (Cell, Storage) =/= true
 -- ------------------------------------------------
 store Data in < Cell > : Action -{storage = Storage,
    storage' = Storage,-}-> failed .

--- 3 (foi dividida em duas)
 store nothing in Yielder2 : Action --> failed .

 store Yielder1 in nothing : Action --> failed .
```

Novamente usamos regras auxiliares para avaliar os *yielders*. Se o primeiro avaliar para um determinado dado (`Data`) e o segundo para uma determinada célula de memória (`< Cell >`) e esta célula existe na memória, uma nova memória é criada por `Storage`' ao amarrar a posição de memória `Cell` ao dado `Data` na memória atual (`Storage`) e o valor de `commitment` é alterado para `true`, conforme foi implementado em `1`. Novamente foi necessário o uso da função de coerção (`< >`), pois como explicado anteriormente dessa forma evitamos o problema de *ad-hoc overloading*. Quando uma posição de memória não existe no *storage* atual ou quando um dos dois *yielders* é avaliado para `nothing`, a ação falha.

```
--- regra auxiliar para avaliar um yielder
            Yielder -{...}-> Yielder'
 -- ---------------------------------------------
```

```
 the DataSort stored in Yielder : Yielder -{...}->
    the DataSort stored in Yielder' .

--- 14 e 15
            Data := lookup (Cell, Storage)
 -- ------------------------------------------------------------
 the DataSort stored in < Cell > : Yielder -{storage = Storage,
    storage' = Storage,-}-> if (Data :< DataSort)
    then Data else nothing fi .

--- 16
            def lookup (Cell, Storage) =/= true
 -- ------------------------------------------------------------
 the DataSort stored in < Cell > : Yielder -{storage = Storage,
    storage' = Storage,-}-> nothing .

--- 17
 the DataSort stored in nothing : Yielder --> nothing .
```

Após avaliar `Yielder`, caso esse produza `nothing` nada é gerado, como é mostrado em `17`. Se uma célula de memória for produzida por esse *yielder* e se ela não existir no *storage* atual, novamente nada é produzido, conforme exibido em `16`. No caso dessa célula existir na memória e o dado por ela armazenado corresponder a `DataSort`, esse dado é produzido. No entanto, se o dado `Data` pertencer a outro *sort*, `nothing` é produzido. Tal comportamento é observado na implementação das transições `14` e `15`.

5.2.2.5 Faceta Reflexiva

A faceta reflexiva foi implementada em três módulos MSDF. O módulo `ReflectiveData` define a assinatura de uma abstração (`Abstraction`), enquanto a sintaxe desta faceta é representada no módulo `ReflectiveSyntax`. Tal sintaxe é usada na definição das suas transições correspondentes em `ReflectiveTransitions`.

Além da sintaxe definida em [Mos99], a qual estamos usando para substituir a Notação de Ações da Semântica de Ações Orientada a Objetos apresentada em [Car02], adicionamos mais um *yielder* à nossa notação, conforme proposto na seção 4.2.2. Esse novo elemento foi chamado de encapsulate e será útil na definição de bibliotecas usando a Semântica de Ações Orientada a Objetos, como é o caso da LFLv2.

A implementação desse *yielder* foi possível a partir da especificação da seguinte transição:

```
encapsulate Action0 : Yielder --> closure abstraction of Action0 .
```

Onde a abstração da ação `Action0` é produzida preservando os *bindings* atuais.

5.2.2.6 Faceta Híbrida

A faceta híbrida é dada pelo uso de mais de uma faceta, por isso é caracterizada por além de criar nova sintaxe, usar a sintaxe e os rótulos de outras facetas existentes. Na implementação de MOOAT apenas um combinador de ações e uma ação foram implementados, são eles: else e allocate Yielder.

```
Action1 else Action2 : Action -->
   (check given truth-value and then Action1) or
   (check not given truth-value and then Action2) .
```

O combinador de ações `else` faz a escolha entre duas ações. Supõe-se que existe um `truth-value` entre os dados transitórios. Esse valor é verificado, caso verdadeiro a ação `Action1` é computada e caso contrário a ação `Action2` é computada. Note que fizemos uso das facetas básica e funcional.

```
       Cell := new-cell (Storage),
       Storage' := (Cell |-> uninitialized) / Storage
-- ------------------------------------------------------------
allocate a cell : Action -{storage = Storage, storage' = Storage',
   commitment = Bool, commitment' = true,-}-> gave (< Cell >) .
```

Para efetuar a alocação de uma célula de memória é preciso reservá-la no *storage*. A partir de `new-cell (Storage)` sabemos qual endereço podemos usar para que a nova memória (`Storage`') seja criada contendo a célula de memória não inicializada e as demais posições já existentes. Além do rótulo especificar o novo *storage*, ele também altera o valor de `commitment` para `true`. A ação é finalizada ao processarmos a nova célula como um dado transitório, para que ela possa ser usada pelas próximas ações. Repare que usamos as facetas imperativa e funcional.

5.2.3 A classe *State*

Em MOOAT a classe *State* foi representada parcialmente por um módulo de sistema chamado `STATE` e parcialmente pela Notação de Classes, a ser e detalhada na próxima seção. Lá veremos também que todas as classes de MOOAT são subclasses diretas de `STATE`.

Já que as classes de MOOAT são transformadas para módulos de sistema em Maude, optamos por implementar a classe `STATE` diretamente como tal. Veremos que outra motivação para essa decisão é o fato de que todas as classes, transformadas em módulos, descendem de `STATE`, já que é esse módulo que agrupa a Notação de Ações apresentada na seção anterior.

A classe `STATE` de MOOAT possui cinco atributos, os quais são usados nas especificações de linguagens. Tais atributos são representados pelos índices dos rótulos implementados pela Notação de Ações descrita anteriormente, sendo:

- `commitment` - é o valor de *commit*;
- `unfolding` - é uma ação;
- `data` - são os dados transitórios (*transients*);
- `bindings` - são os *bindings*;
- `storage` - é a memória (*storage*).

Como trata-se de uma implementação usando MMT, esses atributos devem ser inicializados para que seja possível criar a configuração necessária para computar uma determinada ação. Tal configuração é representada pelo comando `compute`, o qual recebe uma ação como parâmetro e inicializa a configuração MRS usada pela MMT para criar o ambiente de execução que computa a ação.

Os atributos de *State* são inicializados de acordo com o tipo de dado que foi especificado para cada um deles em suas respectivas definições de rótulos, como é mostrado a seguir:

- `commitment` - false;

- `unfolding` - fail;
- `data` - mapeamento vazio (`Int` → `Datum`);
- `bindings` - mapeamento vazio (`Token` → `Data`);
- `storage` - mapeamento vazio (`Cell` → `Data`).

Em MOOAT as operações sobre a classe *State* são as transições implementadas em cada uma das facetas mencionadas na seção anterior, sendo assim as ações da Semântica de Ações Orientada a Objetos de MOOAT são bastante similares as ações especificadas pela MSOS para a Notação de Ações descrita em [Mos99], conforme foi idealizado e proposto por Carvilhe como trabalho futuro em [Car02]. Na próxima seção veremos as peculiaridades em relação ao ambiente dos métodos em MOOAT.

5.2.4 Notação de Classes

A técnica emprega no desenvolvimento da Notação de Classes é similar a usada na construção dos módulos orientados a objetos em Full-Maude [DM99] e na definição de módulos MSDF em MMT [dR05], ou seja, é criada um sintaxe e quando esta é usada suas construções são transformadas em um código que Maude é capaz de interpretar.

Assim como nos exemplos citados como referência, em MOOAT os blocos reutilizáveis, representados por classes da Semântica de Ações Orientada a Objetos, são transformados em módulos de sistema. Para que isso fosse possível, desenvolvemos MOOAT como uma extensão de MMT, a qual é uma extensão de Full-Maude.

As classes em MOOAT são compostas basicamente de três partes: definição de subclasses, definição sintática e definição semântica. Sendo que o uso de cada uma dessas partes pode ser opcional ou seqüencial, ou seja, elas podem não ser usadas, podem ser usadas apenas uma vez ou então podem ser usadas diversas vezes. No entanto, uma classe deve ser composta de pelo menos uma dessas três partes, não sendo permitida a definição de uma classe vazia.

Uma classe em MOOAT é implementada como um módulo MSDF alternativo, onde apenas aquelas três partes citadas anteriormente são aceitas em sua definição. Os módulos

`MSDF-SIGNATURE` e `MSOS-UNIT` foram alterados para que isso fosse possível. A implementação das três partes que compõem uma classe em MOOAT também foi definida no segundo módulo citado.

A definição de subclasses é traduzida para linhas que usam a diretiva `including`. Na parte sintática aproveitamos a definição de árvores sintáticas no estilo BNF implementado pela MMT; essas árvores e os *sorts* sintáticos são transformados, respectivamente, em operações, *sorts* e *subsorts* de Maude. Os métodos da parte semântica são transformados em um conjunto de equações suportado pelos módulos de sistema em Maude, enquanto os objetos definidos automaticamente são tratados como meta-variáveis de Maude nesse conjunto de equações.

Assim como na Semântica de Ações Orientada a Objetos, em MOOAT toda classe é uma subclasse direta de `STATE`, ou seja, sempre que criamos uma nova classe ela é automaticamente incluída para que seja possível acessar a Notação de Ações definida. Por causa da conversão das classes em módulos de sistema, a classe `STATE` foi implementada diretamente como um módulo de sistema. Pois assim é possível que ela seja incluída em todas as classes que são convertidas em MOOAT, já que a conversão é implementada no módulo `MSOS-CONVERTER` pela equação `convertMOOAT`, o qual é definido antes da Notação de Ações e também de `STATE`.

O ambiente de métodos definido pela Semântica de Ações Orientada a Objetos é criado pela inclusão da conversão das classes MOOAT na base de módulos de Maude, dessa forma é implementada a parte faltante de *State*. A classe principal de MOOAT disponibiliza as operações necessárias para alterar os seus atributos, definidos na seção 5.2.3, e o ambiente para a definição de métodos, o qual continua sendo apenas lido pelas operações anteriores, sendo que sempre que uma nova classe for adicionada, os seus respectivos métodos também são adicionados.

A inclusão dos módulos resultantes das conversões de classes, na base de dados de Maude, foi implementada no módulo `MMT-DATABASE-HANDLING`, conforme o seguinte trecho de código:

```
 rl < O : X@Database | db : DB, input : ('class_is_endclass[T, T']),
step-flag : B, output : nil, default : MN, Atts >
    =>
    < O : X@Database | db : mooat-proc-unit('class_is_endclass[T, T'],
step-flag(B), DB), input : nilTermList, step-flag : B,
                   output : ('Introduced 'OOAS 'class
                        header2QidList(parseHeader(T)) '\n),
                   default : parseHeader(T), Atts > .
```

Repare que a inclusão de uma classe na base de módulos é iniciada pela equação `mooat-proc-unit`, a qual foi implementada no módulo `MSOS-PARSER`, que por sua vez implementa as regras necessárias para a análise sintática de uma classe. O processo é finalizado quando o controle é passado para a equação `mooat-eval-preunit`, do módulo `MSOS-SOLVER`, para que a classe seja realmente convertida em módulo de sistema e em seguida este seja adicionado a base de dados presente no ambiente de Maude.

5.3 Considerações finais

A primeira implementação da Semântica de Ações Orientada a Objetos foi apresentada neste capítulo. Explicamos a notação e o processo de desenvolvimento da ferramenta. Quanto a notação, ela é bastante similar a notação proposta por Carvilhe [Car02]. No entanto, algumas adaptações foram feitas devido a algumas peculiaridades do interpretador Maude.

A implementação consiste de uma adaptação das regras SOS, usadas para definir a Notação de Ações da Semântica de Ações Orientada a Objetos [Car02], para regras MSOS introduzidas por Mosses [Mos04] e implementadas pela MMT [dR05]. A Notação de Classes foi implementada usando o sistema disponibilizado por Maude.

MOOAT colocou na prática as idéias propostas por Carvilhe [Car02], com a intenção de prover uma ferramenta e um ambiente executável para especificar linguagens usando a Semântica de Ações Orientada a Objetos. Já que nenhuma ferramenta havia sido desenvolvida até então [vdBIM06]. Dessa forma, a ferramenta pode ser utilizada para lecionar os conceitos da Semântica de Ações Orientada a Objetos, ou até mesmo da Semântica de Ações pura.

CAPÍTULO 6

ESTUDOS DE CASO

Neste capítulo apresentaremos três estudos de caso desenvolvidos em MOOAT. O primeiro deles é uma linguagem imperativa simples chamada μ-Pascal. O segundo é a implementação da LFL versão 2 em MOOAT e a demonstração do uso da biblioteca para especificar μ-Pascal. O terceiro estudo de caso trata da criação da Semântica de Ações Orientada a Objetos Construtiva e como esta pode ser usada para especificar μ-Pascal. Finalizando o capítulo, apresentamos um comparativo entre o uso da biblioteca e da abordagem construtiva na Semântica de Ações Orientada a Objetos.

6.1 μ-Pascal + OOAS + MOOAT

Nesta seção apresentaremos a especificação de uma linguagem chamada μ-Pascal. Esta linguagem é bastante similar à linguagem Pascal. μ-Pascal é uma linguagem de programação imperativa que contém basicamente comandos e expressões. A linguagem μ-Pascal será especificada usando a ferramenta descrita no capítulo anterior.

```
(class Identifier is
 Id .
 Id ::= @ Token .
 absmethod convert Id -> Token .
 method convert (@ T:Token) = T:Token .
endclass)
```

A classe `Identifier` representa os identificadores aceitos pela linguagem pelo uso do *sort* `Id`. Na Notação de Ações foi especificado que um `Token` é um elemento do *sort* `Qid`, no entanto não podemos fazer com que `Token` seja um *subsort* direto de `Id`, pois dessa forma teríamos o problema de *ad-hoc overloading* citado na Seção 3.1.3. Por isso usamos o caractere arroba (`@`) como função de coerção para a definição dos identificadores usados na linguagem. O método abstrato `convert` é declarado para indicar a assinatura

da conversão de um identificador em um *token* e em seguida é implementado. O uso de tal método será necessário nas ações onde um identificador deve ser convertido em um *token* para que elas sejam computadas.

```
(class Declaration is
 Dec .
 absmethod elaborate Dec -> Action .
endclass)

(class Command is
 Cmd .
 absmethod execute Cmd -> Action .
endclass)

(class Expression is
 Exp .
 absmethod evaluate Exp -> Action .
endclass)
```

As três classes abstratas definidas representam as definições de declarações, comandos e expressões, respectivamente. Cada uma delas declara o *sort* e o método abstrato que serão sobrecarregados nas classes especializadas para especificar uma determinada característica da linguagem, bem próximo da notação usada na Semântica de Ações Orientada a Objetos.

```
(class ConstantDec is
 extends Declaration .
 Dec ::= const Id = Exp .
 method elaborate (const I:Id = E:Exp) = (evaluate E:Exp)
    then (bind < convert I:Id > to given value) .
endclass)
```

Pelo uso da diretiva `extends` definimos que a classe `ConstantDec` é uma subclasse de `Declaration`. Na parte sintática redefinimos o *sort* `Dec`, especificando a estrutura de uma declaração de constante. A semântica dessa declaração é dada pelo método `elaborate` implementado, o qual faz uso dos objetos `I:Id` e `E:Exp` para efetuar *binding* do identificador `I` ao valor produzido pela chamada do método `evaluate` do objeto `E`. Vale lembrar que na ferramenta a declaração dos objetos é automática e agora na sua declaração é usado o *sort* do objeto, como na especificação da sintaxe, em vez da superclasse, como era usado na Semântica de Ações Orientada a Objetos. Vimos também o

uso do método `convert`, já que um identificador deve ser convertido em um *token* usado por MOOAT.

```
(class VariableDec is
 extends Declaration .
 Dec ::= var Id .
 Dec ::= var Id = Exp .
 method elaborate (var I:Id) = (allocate a cell)
    then (bind < convert I:Id > to given cell) .
 method elaborate (var I:Id = E:Exp) =
    ((evaluate E:Exp) and (allocate a cell))
    then ((bind < convert I:Id > to given cell)
    and store given value in given cell) .
endclass)
```

A classe `VariableDec` é mais uma subclasse de `Declaration`. Nela as estruturas de uma declaração de variável são definidas a partir da redefinição de `Dec`. Essas duas possibilidades de declaração de variáveis são implementadas pela sobrecarga do método `elaborate`. Primeiramente é definida uma declaração de variável sem inicialização e posteriormente é contemplada a inicialização da variável declarada. No primeiro caso, uma célula de memória é alocada e amarrada ao *token* representado pelo identificador `I`. Na segunda implementação de `elaborate` o *binding* é representado da mesma maneira; contudo, um valor, resultante da chamada do método `evaluate` do objeto `E`, é armazenado na posição de memória reservada

```
(class DecSeq is
 extends Declaration .
 Dec ::= Dec ; Dec .
 method elaborate (D1:Dec ; D2:Dec) =
    elaborate D1:Dec before elaborate D2:Dec .
endclass)
```

A seqüenciação de declarações é dada na classe `DecSeq` pela redefinição do método `elaborate`, onde as declarações representadas pelos objetos `D1` e `D2` são processadas de acordo com a sintaxe redefinida por `Dec`.

```
(class Assign is
 extends Command .
 Cmd ::= Id := Exp .
 method execute (I:Id := E:Exp) = (evaluate E:Exp)
```

```
    then store given value in the cell bound to < convert I:Id > .
endclass)
```

O comando de atribuição é implementado na classe `Assign`, a qual é uma subclasse de `Command`. O *sort* `Cmd` define a estrutura sintática de uma atribuição, enquanto o método `execute` expressa a sua respectiva semântica usando os objetos `I:Id` e `E:Exp`. A ação deste método armazena o valor resultante da avaliação do objeto `E` na célula de memória correspondente a `I`.

```
(class Selection is
 extends Command .
 Cmd ::= if Exp then Cmd end-if .
 Cmd ::= if Exp then Cmd else Cmd end-if .
 method execute (if E:Exp then C1:Cmd end-if) =
    (evaluate E:Exp) then
    (execute C1:Cmd else complete) .
 method execute (if E:Exp then C1:Cmd else C2:Cmd end-if) =
    (evaluate E:Exp) then
    (execute C1:Cmd else execute C2:Cmd) .
endclass)
```

Na classe `Selection` especificamos as duas estruturas possíveis de um comando de seleção implementado, por isso o método `execute` foi redefinido duas vezes. Em ambos os métodos a expressão representada pelo objeto `E` é avaliada e produz um valor verdade. No primeiro método, existe apenas a possibilidade do comando `C1` ser executado cujo o valor da expressão seja verdadeiro. Já no segundo método, existe a possibilidade de execução do comando `C1` em caso de verdade e a execução de `C2` em caso de falsidade.

```
(class Repeat is
 extends Command .
 Cmd ::= repeat Cmd until Exp .
 method execute (repeat C:Cmd until E:Exp) = unfolding ((execute C:Cmd)
    and then ((evaluate E:Exp) then (unfold else complete))) .
endclass)
```

O laço de repetição da linguagem μ-Pascal é implementado na classe `Repeat`. Assim como nas outras subclasses de `Command`, a sintaxe do comando representado pela classe especializada é dada pela sobrecarga de `Cmd`. Os objetos `C:Cmd` e `E:Exp` são usados na redefinição do método execute. A semântica da estrutura de repetição é dada pela execução

do comando C até a expressão E deixar de ser verdadeira.

```
(class Sequence is
 extends Command .
 Cmd ::= Cmd ; Cmd .
 method execute (C1:Cmd ; C2:Cmd) =
    (execute C1:Cmd and then execute C2:Cmd) .
endclass)
```

A seqüenciação de comandos é expressa na classe `Sequence` ao redefinirmos o método `execute` e o *sort* `Cmd`. Os dois comandos são representados pelos objetos `C1` e `C2`, respectivamente, sendo que o segundo comando é executado apenas quando o primeiro terminar completamente a sua execução.

```
(class Number is
 extends Expression .
 Exp ::= Int .
 method evaluate (N:Int) = give < N:Int > .
endclass)
```

A classe `Number` representa a semântica de um número em expressões, por isso é uma subclasse de `Expression`. O objeto `N:Int` é usado, já que definimos que os números usados em μ-Pascal são inteiros. A semântica do método `evaluate` consiste na produção do número inteiro representado por `N`.

```
(class IdExp is
 extends Expression .
 Exp ::= Id .
 method evaluate (I:Id) = give (the value bound to < convert I:Id >) or
    give (the value stored in the cell bound to < convert I:Id >) .
endclass)
```

Um identificador pode ser uma variável ou uma constante, como definimos nas declarações. A semântica de um identificador é dada pela classe `IdExp`. O *sort* `Exp` é redefinido com a estrutura sintática de um identificador (`Id`) e o método `evaluate` é sobrecarregado para definir a semântica da avaliação de um identificador, a qual produz o valor amarrado ao identificador `I` ou o valor armazenado na célula de memória amarrada ao identificador `I`.

```
(class Sum is
```

```
 extends Expression .
 Exp ::= Exp '+ Exp .
 method evaluate (E1:Exp '+ E2:Exp) =
    (evaluate E1:Exp and evaluate E2:Exp)
    then give sum(given integer # 1, given integer # 2) .
endclass)

(class Sub is
 extends Expression .
 Exp ::= Exp '- Exp .
 method evaluate (E1:Exp '- E2:Exp) =
    (evaluate E1:Exp and evaluate E2:Exp)
    then give difference(given integer # 1, given integer # 2) .
endclass)

(class Prod is
 extends Expression .
 Exp ::= Exp '* Exp .
 method evaluate (E1:Exp '* E2:Exp) =
    (evaluate E1:Exp and evaluate E2:Exp)
    then give product(given integer # 1, given integer # 2) .
endclass)
```

A definição das classe `Sum`, `Sub` e `Prod` segue uma estrutura similar. A diferença é observada na definição da sintaxe, onde definimos os operadores binários `'+`, `'-` e `'*` para as operações de soma, subtração e multiplicação, respectivamente. Note que foi necessário usar aspas simples (') antes da definição da operação, pois caso contrário recairíamos no problema de *ad-hoc overloading*, já que esses operadores já são utilizados por Maude e não poderiam ser sobrecarregados. A semântica dessas operações aritméticas é dada pela sobrecarga do método `evaluate` e nas três classes são usados os objetos `E1:Exp` e `E2:Exp` para representar as duas expressões que estão sendo avaliadas. O resultado final é dado ao empregarmos as funções auxiliares `sum(_,_)`, `difference(_,_)` e `product(_,_)` para efetuar respectivamente: soma, subtração e multiplicação nos valores das expressões.

```
(class True is
 extends Expression .
 Exp ::= true .
 method evaluate (true) = give < tt > .
endclass)

(class False is
 extends Expression .
```

```
 Exp ::= false .
 method evaluate (false) = give < ff > .
endclass)
```

As classe `True` e `False` definem a semântica das expressões representadas pelos terminais `true` e `false`. Em ambas as classes o método `evaluate` é sobrecarregado para implementar a ação que produz o valor correspondente a cada uma das frases.

```
(class LessThan is
 extends Expression .
 Exp ::= Exp '< Exp .
 method evaluate (E1:Exp '< E2:Exp) =
    (evaluate E1:Exp and evaluate E2:Exp)
    then give less(given integer # 1, given integer # 2) .
endclass)

(class Equality is
 extends Expression .
 Exp ::= Exp '= Exp .
 method evaluate (E1:Exp '= E2:Exp) =
    (evaluate E1:Exp and evaluate E2:Exp)
    then give equal(given integer # 1, given integer # 2) .
endclass)
```

As expressões lógicas, representadas pelos operadores `'<` e `'=`, são definidas nas classes `LessThan` e `Equality`, respectivamente. A estrutura dessas classes é similar a definida pelas classes das expressões aritméticas (ex: `Sum`). As funções auxiliares `less(_,_)` e `equal(_,_)` definem, respectivamente, a verificação se uma expressão é menor que outra e a verificação da igualdade entre duas expressões.

```
(class Micro-Pascal is
 Prog .
 Prog ::= begin Dec ; Cmd end .
 absmethod run Prog -> Action .
 method run (begin D:Dec ; C:Cmd end) =
    elaborate D:Dec hence execute C:Cmd .
endclass)
```

A linguagem pode ser representada pela classe `Micro-Pascal`, onde definimos a estrutura de um programa através da criação e definição do *sort* `Prog`. O método `run` é declarado para que a semântica de um programa seja representado por uma ação. A

semântica em si é dada pela elaboração do objeto `D:Dec` e em seguida pela execução de `C:Cmd`.

```
(class Factorial is
 extends ConstantDec, VariableDec, DecSeq .
 extends Assign, Selection, Repeat, Sequence .
 extends Number, IdExp, Sum, Sub, Prod .
 extends True, False, LessThan, Equality .
 extends Micro-Pascal .

 absmethod fat -> Action .
 method fat = (run
            (begin
                ((var @ 'a = 1) ;
                (var @ 'b = 0)) ;
                (repeat
                    ((@ 'a := (@ 'a '+ (@ 'a '* @ 'b))) ;
                    (@ 'b := (@ 'b '+ 1)))
                until (@ 'b '< 5))
            end)
            ) .
endclass)
```

Como vimos na seção 5.1.4, precisamos criar uma classe onde seja possível acessar todos os métodos definidos anteriormente para que seja possível executar um programa, definido na linguagem em questão. Aqui, a classe `Factorial` é responsável por isso e além de estender as classes no nível mais baixo da hierarquia da linguagem, também cria o método `fat`, o qual é a representação de um programa para o cálculo do fatorial de cinco na linguagem μ-Pascal, especificada em MOOAT.

6.2 LFLv2 + MOOAT

Nesta seção mostraremos como a LFL versão 2, apresentada no capítulo 4, foi implementada em MOOAT. Apresentaremos apenas algumas classes, necessárias para a compreensão da implementação da nova LFL. Todas as classes implementadas podem ser encontradas no Apêndice D, o qual prova as novas características da biblioteca.

```
(class LFL is
 extends STATE .
endclass)
```

A classe `LFL` não existia na versão formal da biblioteca, mas ela foi criada na versão implementada para que ficasse mais clara a hierarquia por ela criada. Sendo uma classe da Semântica de Ações Orientada a Objetos implementada por MOOAT, não seria preciso especificar que esta é uma subclasse de `STATE`, mas como não podemos criar classes vazias explicitamos a derivação para que fosse possível criar a classe em questão.

```
(class LFL.Declaration is
 extends LFL .
endclass)

(class LFL.Declaration.Paradigm is
 extends LFL.Declaration .
endclass)

(class LFL.Declaration.Paradigm.Imper is
 extends LFL.Declaration.Paradigm .
endclass)
```

Na implementação da LFL, em vez de usarmos a diretiva `locating`, presente na ferramenta formal, usaremos o possível conteúdo dessa diretiva como o nome das classes.

Assim como a classe `LFL`, as classes `LFL.Declaration`, `LFL.Declaration.Paradigm` e `LFL.Declaration.Paradigm.Imper` não existiam no método formal. No entanto, foram implementadas para que a hierarquia da LFL ficasse clara. Note que a hierarquia usada na implementação é a mesma proposta na seção 4.2.1, com ressalvas às peculiaridades da ferramenta usada.

A seguir apresentaremos uma classe de cada um dos três nós principais da biblioteca, pois julgamos suficiente a explicação dessas classes para o entendimento das restantes. A primeira será relacionada ao nodo *Declaration*, em seguida apresentaremos uma classe do nodo *Command* e por fim alguma de *Expression*.

```
(class LFL.Declaration.Paradigm.Imper.Variable is
 extends LFL.Declaration.Paradigm.Imper .
 absmethod elaborate-variable (Token, Yielder) -> Action .
 method elaborate-variable (T:Token, Y:Yielder) =
    ((enact Y:Yielder) and (allocate a cell))
    then ((bind < T:Token > to given cell) and
    store given value in given cell) .
endclass)
```

A classe `LFL.Declaration.Paradigm.Imper.Variable` representa a implementação da classe `Variable`, descrita na seção 4.3.1.2. Note que aqui explicitamos que esta é uma subclasse de `LFL.Declaration.Paradigm.Imper`. O método `elaborate-variable` precisa antes ser declarado abstrato porque como as classes são transformadas em módulos de sistema, não é possível criar uma equação sem antes definir a sua assinatura através de uma operação, da mesma forma como foi visto na seção 5.2.

O método `elaborate-variable` foi implementado de forma pouco diferente da apresentada na seção 4.3.1.2. Em vez dele receber três parâmetros (`Token`, `Sort`, `Yielder`) apenas dois são usados (`Token`, `Yielder`), devido ao fato de que a alocação de variável atrelada ao tipo de dado usado não foi implementado em MOOAT. Quanto ao significado semântico deste método, a única diferença é relacionada a alocação da posição de memória citada.

```
(class LFL.Command.Shared.Selection is
 extends LFL.Command.Shared .
 absmethod execute-if-then (Yielder, Yielder) -> Action .
 absmethod execute-if-then-else (Yielder, Yielder, Yielder) -> Action .
 method execute-if-then (Y1:Yielder, Y2:Yielder) =
    (enact Y1:Yielder) then (enact Y2:Yielder else complete) .
 method execute-if-then-else (Y1:Yielder, Y2:Yielder, Y3:Yielder) =
    (enact Y1:Yielder) then (enact Y2:Yielder else enact Y3:Yielder) .
endclass)
```

A classe `LFL.Command.Shared.Selection` é a implementação da classe `Selection`, apresentada na seção 4.3.2.4. Assim como a classe formal, a implementada está ligada diretamente ao nodo `LFL.Command.Shared` e implementa os métodos `execute-if-then` e `execute-if-then-else` da mesma forma como foi proposto na reestruturação da biblioteca. Lembramos apenas que em MOOAT é preciso definir os métodos abstratos antes da definição dos métodos propriamente ditos, e a declaração dos objetos é dada de forma automática na definição desses métodos.

```
(class LFL.Expression.Shared.Sum is
 extends LFL.Expression.Shared .
 absmethod evaluate-sum (Yielder, Yielder) -> Action .
 method evaluate-sum (Y1:Yielder, Y2:Yielder) =
    (enact Y1:Yielder and enact Y2:Yielder) then
    give sum(given integer # 1, given integer # 2) .
```

```
endclass)
```

A classe `Sum`, da seção 4.3.3.1, é implementada por `LFL.Expression.Shared.Sum`. É uma classe pertencente ao nodo `LFL.Expression.Shared` e implementa o método `evaluate-sum` como foi definido pela LFL versão 2, além de seguir as restrições impostas pela ferramenta descrita por este trabalho.

As demais classes implementadas estão descritas no Apêndice D. A compreensão do conteúdo do apêndice mencionado é possível por uma simples comparação entre a classes formais e as classes implementadas, conforme explicado nesta seção. Visto que a implementação segue a risca os significados semânticos propostos pelas classes da seção 4.3.

6.2.1 μ-Pascal + LFLv2 + MOOAT

Apresentaremos de forma sucinta como usar a LFL, implementada na seção anterior, na linguagem μ-Pascal, definida na seção 6.1. Gostaríamos de ressaltar que a biblioteca permite a definição da linguagem de forma independente à sua sintaxe, pois a definição semântica está relacionada ao uso dos métodos disponibilizados pela biblioteca, os quais podem ser usados na definição de várias linguagens, não importando a sintaxe dela.

```
(class VariableDec is
 extends Declaration .
 extends LFL.Declaration.Paradigm.Imper.Variable .
 extends LFL.Declaration.Paradigm.Imper.VariableDec .
 Dec ::= var Id .
 Dec ::= var Id = Exp .
 method elaborate (var I:Id) =
    elaborate-variable (convert I:Id) .
 method elaborate (var I:Id = E:Exp) =
    elaborate-variable (convert I:Id, encapsulate (evaluate E:Exp)) .
endclass)
```

A classe `VariableDec` é similar a definida na seção 6.1. Entretanto, ao invés de definirmos o significado semântico usando a Notação de Ações, disponibilizada pela Semântica de Ações Orientada a Objetos, usamos os métodos `elaborate-variable` definidos nas suas respectivas classes da biblioteca.

Além disso, diferentemente da versão formal da biblioteca, onde era instânciado um objeto para acessar os métodos de uma determinada classe, agora incluimos a classe que define os métodos necessários na classe em que precisamos usá-los.

```
(class Selection is
 extends Command .
 extends LFL.Command.Shared.Selection .
 Cmd ::= if Exp then Cmd end-if .
 Cmd ::= if Exp then Cmd else Cmd end-if .
 method execute (if E:Exp then C1:Cmd end-if) =
    execute-if-then (encapsulate (evaluate E:Exp),
                     encapsulate (execute C1:Cmd)) .
 method execute (if E:Exp then C1:Cmd else C2:Cmd end-if) =
    execute-if-then-else (encapsulate (evaluate E:Exp),
    encapsulate (execute C1:Cmd), encapsulate (execute C2:Cmd)) .
endclass)
```

A classe `Selection` é semelhante a definida na seção 6.1. No entanto, aqui estendemos a classe `LFl.Command.Shared.Selection` para que fosse possível acessar os métodos `execute-if-then` e `execute-if-then-else`. Os parâmetros desses métodos são criados com ajuda do novo *yielder* `encapsulate`, o qual cria uma abstração de ação necessária para a computação da mesma em outro ponto do código, independentemente dos métodos definidos pelo usuário.

```
(class Sum is
 extends Expression .
 extends LFL.Expression.Shared.Sum .
 Exp ::= Exp '+ Exp .
 method evaluate (E1:Exp '+ E2:Exp) =
    evaluate-sum (encapsulate (evaluate E1:Exp),
                  encapsulate (evaluate E2:Exp)) .
endclass)
```

A classe `Sum` é similar a da seção 6.1, e assim como as demais classes apresentadas nesta seção, faz uso de um determinado método implementado por uma determinada classe da biblioteca de classes implementada em MOOAT. Aqui usamos o método `evaluate-sum` para implementar a soma de duas expressões, independentemente da sintaxe usada na linguagem em que estamos descrevendo. Voltamos a informar que isso foi possível graças ao uso das abstrações de ações criadas pela diretiva `encapsulate`, as quais são execu-

tadas pela biblioteca para especificar o comportamento semântico de uma determinada característica da linguagem que está sendo definida.

Apresentamos nesta seção apenas algumas classes para a compreensão do uso da nova LFL em MOOAT, as quais são suficientes para a compreensão das classes descritas no Apêndice D.1. Tal implementação prova as características da LFL, onde ressaltamos que a mais importante é a definição de linguagens independentemente de sintaxe, pelo uso de métodos disponibilizados por uma biblioteca de classes.

6.3 COOAS + MOOAT

Nesta seção apresentaremos a Semântica de Ações Orientada a Objetos Construtiva, ou do inglês *Constructive Object-Oriented Action Semantics* (COOAS). Trata-se da junção dos conceitos propostos por [Car02] com a abordagem apresentada em [Mos05]. Apresentaremos apenas algumas classes para a compreensão da implementação, as demais podem ser encontradas no Apêndice E.

Usando MOOAT veremos como a abordagem construtiva de Mosses pode ser aplicada na Semântica de Ações Orientada a Objetos para criar especificações de linguagens independentemente de sintaxe e também como uma alternativa ao uso da LFL.

```
(class Exp is
 Exp .
 absmethod evaluate Exp -> Action .
endclass)
```

A classe abstrata `Exp` é responsável por introduzir o *sort* sintático `Exp`, o qual será usado nas definições das construções de expressões implementadas. Também é introduzido o método abstrato `evaluate` para funcionar como uma função semântica.

```
(class Exp/Val is
 extends Exp .
 Value .
 Value ::= Int .
 Value ::= Boolean .
 Exp ::= Value .
 method evaluate (V:Value) = give < V:Value > .
endclass)
```

Na classe `Exp/Val` definimos que os valores, representados pelo *sort* `Value`, são *subsorts* de expressões e por isso ela é uma subclasse de `Exp`. O objeto `V:Value` é automaticamente declarado para representar a semântica de um valor usando a Notação de Ações disponibilizada por MOOAT.

```
(class Exp/Val-Id is
 extends Exp .
 Exp ::= val (Token) .
 method evaluate (val(T:Token)) =
    give (the value bound to < T:Token >) or
    give (the value stored in the cell bound to < T:Token >) .
endclass)
```

O construtor `val(T:Token)` é definido pela classe `Exp/Val-Id`. Tal construtor é responsável por representar a avaliação de identificadores. Um identificador é representado pelo objeto `T:Token`, que na sua avaliação resulta em um valor amarrado a uma variável `T` ou em um valor armazenado em uma posição de memória atrelada a variável `T`.

```
(class Exp/Sum is
 extends Exp .
 Exp ::= exp-sum (Exp, Exp) .
 method evaluate (exp-sum(E1:Exp, E2:Exp)) =
    (evaluate E1:Exp and evaluate E2:Exp) then
    give sum(given integer # 1, given integer # 2) .
endclass)
```

Uma soma de expressões é definida pelo construtor, `exp-sum(Exp, Exp)`, implementado na classe `Exp/Sum`. A soma é dada pela aplicação do método auxiliar `sum(_,_)` sobre dois números inteiros, resultantes da avaliação das expressões contidas nos objetos `E1:Exp` e `E2:Exp`.

```
(class Cmd is
 Cmd .
 absmethod execute Cmd -> Action .
endclass)
```

Assim como para as expressões, também definimos uma classe abstrata para introduzir o *sort* sintático dos construtores de comandos. Essa classe é representada por `Cmd`. A função semântica `execute` é definida por meio da declaração de um método abstrato que faz o mapeamento do sort sintático `Cmd` para uma ação.

```
(class Cmd/Repeat is
 extends Cmd .
 Cmd ::= cmd-repeat (Cmd, Exp) .
 method execute (cmd-repeat(C:Cmd, E:Exp)) =
    unfolding ((execute C:Cmd and then evaluate E:Exp)
    then (unfold else complete)) .
endclass)
```

A classe `Cmd/Repeat` é uma classe especializada de `Cmd` e implementa o construtor de um laço de repetição, `cmd-repeat(Cmd, Exp)`. A iteração no construtor do laço de repetição é executada pelas ações `unfolding A` e `unfold`. A ação `unfold` executa a ação `A`, resultante da última ocorrência de `unfolding A`. Os objetos `C:Cmd` e `E:Exp` são usados para representar o comando a ser executado e a expressão que determina o critério de parada do laço.

```
(class Cmd/Sequence is
 extends Cmd .
 Cmd ::= cmd-seq (Cmd, Cmd) .
 method execute (cmd-seq(C1:Cmd, C2:Cmd)) =
    execute C1:Cmd and then execute C2:Cmd .
endclass)
```

A seqüenciação de comandos é representada pelo construtor `cmd-seq(Cmd,Cmd)`, implementado na classe `Cmd/Sequence`. Os objetos `C1:Cmd` e `C2:Cmd` determinam os comandos a serem executados, de tal forma que o segundo é executado apenas quando a execução do primeiro termina completamente.

```
(class Dec is
 Dec .
 absmethod elaborate Dec -> Action .
endclass)
```

A classe `Dec` introduz o *sort* sintático de mesmo nome a ser usado para representar as construções das declarações. Da mesma forma como nas expressões e nos comandos, esta é uma classe abstrata que implementa um método abstrato para ser usado nas classes especializadas.

```
(class Dec/Variable is
 extends Dec .
```

```
 Dec ::= dec-var (Token, Exp) .
 method elaborate (dec-var(T:Token, E:Exp)) =
    (evaluate E:Exp and allocate a cell)
    then ((bind < T:Token > to given cell)
    and (store given value in given cell)) .
endclass)
```

O construtor `dec-var(Token, Exp)` representa uma declaração de variável com atribuição implementada pela classe `Dec/Variable`, a qual é uma subclasse de `Dec`. O método `elaborate` é sobrecarregado para especificar o comportamento semântico por meio da Semântica de Ações Orientada a Objetos implementada por MOOAT. Sendo assim, a avaliação da expressão representada pelo objeto `E:Exp` resulta em um valor a ser armazenado em uma posição de memória alocada e amarrada ao identificador contido no objeto `T:Token`.

```
(class Prog is
 Prog .
 Prog ::= prog (Dec, Cmd) .
 absmethod run Prog -> Action .
 method run (prog (D:Dec, C:Cmd)) =
    elaborate D:Dec hence execute C:Cmd .
endclass)
```

A construção de um programa pode ser representada por `prog(Dec,Cmd)`, ou seja, declarações seguidas de comandos. O método `run` descreve a semântica de forma que os possíveis comandos contidos no objeto `C:Cmd` são executados após a performance das possíveis declarações representadas pelo objeto `D:Dec`. Essas características foram implementadas pela classe `Prog`.

6.3.1 μ-Pascal + COOAS + MOOAT

A seguir apresentaremos como a linguagem μ-Pascal, especificada na seção 6.1, pode ser definida usando a Semântica de Ações Orientada Objetos Construtiva, especificada em MOOAT e apresentada na seção anterior.

Gostaríamos de ressaltar que a Semântica de Ações Orientada a Objetos Construtiva permite a definição de linguagens de forma independente de sintaxe graças a imple-

mentação de construtores independentes em classes separadas com o intuito de representar determinados conceitos presentes em linguagens de programação.

```
(class Micro-Pascal is
 extends Exp .
 extends Exp/Val .
 extends Exp/Val-Id .
 extends Exp/Sum .
 extends Exp/Sub .
 extends Exp/Prod .
 extends Exp/True .
 extends Exp/False .
 extends Exp/LessThan .
 extends Exp/Equality .

 extends Cmd .
 extends Cmd/Assignment .
 extends Cmd/Repeat .
 extends Cmd/Sequence .
 extends Cmd/Cond .

 extends Dec .
 extends Dec/Variable .
 extends Dec/DecSeq .

 extends Prog .
endclass)
```

Já que cada construção foi especificada em uma classe separada, a especificação de uma linguagem é obtida pela combinação dessas classes. Por isso criamos a classe `Micro-Pascal` de forma a estender as classes definidas no estilo da Semântica de Ações Orientada a Objetos Construtiva.

Dessa mesma maneira que criamos a especificação da linguagem μ-Pascal poderíamos ter criado especificações para outras linguagens, ou seja, apenas inserindo os construtores necessários na linguagem implementada. Isso garante a especificação independente de sintaxe proposta em [Mos05].

```
(class Factorial is
 extends Micro-Pascal .

 absmethod fat -> Action .
 method fat = (run
```

```
                (prog(
                    dec-seq(dec-var('a, 1),dec-var('b, 0)),
                    cmd-repeat(cmd-seq(
                    cmd-assign('a,exp-sum(val('a),
                               exp-prod(val('a),val('b)))),
                    cmd-assign('b,exp-sum(val('b), 1))),
                    exp-less(val('b), 5)))
                    )
                ) .
endclass)
```

Criamos a classe `Factorial` para exemplificar como seria a descrição de um programa para o cálculo do fatorial de cinco na linguagem em questão. Assim como vimos na seção 5.1.4, precisamos criar uma classe onde seja possível acessar os métodos que permitem a execução de um programa na linguagem. Como a definição da linguagem foi possível em apenas uma classe, apenas essa precisa ser especificada.

Repare que como no exemplo da seção 6.1, o método `fat` é responsável por representar o programa que efetua o cálculo do fatorial. No entanto, o programa é escrito usando os construtores definidos pelas classes especificadas em MOOAT, ao invés de usar a própria sintaxe da linguagem. Isto é uma característica da abordagem construtiva, como foi explicado na seção 2.5 e como foi proposto em [Mos05].

6.4 Comparativo entre a LFLv2 e a COOAS

Nesta seção apresentaremos um breve comparativo entre a nova LFL (LFLv2), apresentada no Capítulo 4 e implementada na Seção 6.2, e a Semântica de Ações Orientada a Objetos Construtiva (COOAS), implementada na Seção 6.3.

Destacamos como principal vantagem da biblioteca em relação a abordagem construtiva a preservação da sintaxe da linguagem, ou seja, após especificarmos uma linguagem de programação usando a LFL e a Semântica de Ações Orientada a Objetos é possível escrever programas para testar a especificação usando a própria sintaxe da linguagem descrita.

Enquanto na Semântica de Ações Orientada a Objetos Construtiva, bem como nas outras variantes da abordagem construtiva, deve-se escrever programas para testar a es-

pecificação usando a sintaxe criada pelos construtores usados na especificação. Sendo assim, é necessário o desenvolvimento de um interpretador para cada linguagem especificada, ou seja, uma nova ferramenta deve ser escrita para traduzir a sintaxe da linguagem na sintaxe dos construtores usados na especificação.

No entanto, o uso da Semântica de Ações Orientada a Objetos Construtiva é de certa forma mais conveniente em relação à Semântica de Ações Orientada a Objetos com a LFL. Pois na primeira abordagem citada é possível construir uma especificação em apenas uma classe, estendendo outras classes que implementam os construtores necessários, fazendo com que a semântica da linguagem seja expressa usando a sintaxe dos construtores, como mencionado anteriormente.

Já na segunda abordagem citada, o número de classes é maior fazendo com que a especificação também seja maior e leve mais tempo para ser construída porque é dessa forma que a sintaxe da linguagem é usada na descrição dos comportamentos semânticos. Nessa abordagem a semântica de um determinado comportamento da linguagem é dada pelo uso de um determinado método implementado pela biblioteca.

Levando em consideração os pontos citados anteriormente, podemos resumir os pontos positivos e negativos de cada abordagem da seguinte maneira:

LFLv2

- (+) Permite a especificação de linguagens independente de sintaxe por meio do reúso de métodos disponibilizados pela biblioteca.
- (+) Preserva a sintaxe da linguagem especificada.
- (+) Programas podem ser escritos usando a sintaxe da linguagem especificada.
- (-) A especificação é de certa forma extensa e dividida em diversas classes.
- (-) A escrita da especificação é mais demorada em relação a COOAS.

COOAS

- (+) Permite a especificação de linguagens independente de sintaxe por meio do reúso de construtores previamente definidos.

- (-) Cria uma nova sintaxe da linguagem especificada de acordo com os construtores usados.
- (-) Para escrever programas usando a sintaxe da linguagem especificada deve ser escrita uma ferramenta que a transforme na sintaxe criada pelos construtores.
- (+) A especificação é compacta e pode ser especificada em apenas uma classe.
- (+) A escrita da especificação é mais rápida em relação a LFLv2.

Como o nosso objetivo é, justamente, a especificação formal de linguagens independente de sintaxe, consideramos o uso da biblioteca mais prático. Visto que é possível especificar uma determinada linguagem sem se preocupar com a sintaxe dela e pelo fato de que é possível escrever programas para testar a especificação usando a própria sintaxe da linguagem que está sendo definida.

CAPÍTULO 7

CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho apresentamos MOOAT (*Maude Object-Oriented Action Tool*), a primeira implementação da Semântica de Ações Orientada a Objetos. Tal implementação consiste na adaptação da Notação de Ações, definida pela Semântica de Ações Orientada a Objetos original por meio de transições SOS [Plo81], em transições MSOS introduzidas por Mosses em [Mos04] e suportadas pela MMT [dR05]. Além disso, a Notação de Classes foi implementada usando o poderoso sistema provido por Maude [CDE+05].

Além da implementação, uma nova versão da LFL foi apresentada com o intuito de resolver os problemas encontrados na primeira versão. Tais problemas foram solucionados pelo uso de *abstrações* de ações ao invés de passar classes e métodos como parâmetros para as classes da LFL. A biblioteca também foi reestruturada para implementar apenas os conceitos relacionados à representação da semântica de linguagens de programação. Nessas considerações, a LFLv2 é mais concisa e simplificada devido ao uso da faceta reflexiva.

Além de definir as modificações necessárias para a criação da LFLv2, a biblioteca foi implementada e testada na ferramenta descrita por este trabalho. Assim como a combinação da Semântica de Ações Orientada a Objetos com a Semântica de Ações Construtiva originou a Semântica de Ações Orientada a Objetos Construtiva como um estudo de caso de MOOAT. Esta última semântica construtiva representa um novo estilo de uso do formalismo orientado a objetos. E é a primeira vez que as duas abordagens foram combinadas.

Tanto a LFLv2 como a Semântica de Ações Orientada a Objetos Construtiva melhoram os aspectos modulares encontrados na abordagem orientada a objetos original. Além de que, ambas as opções podem ser usadas para definir especificações semânticas de linguagens de programação independente de sintaxe. No entanto, cada uma dessas opções

possui suas vantagens e desvantagens, como pode ser observado no breve comparativo traçado entre as duas abordagens e apresentado na Seção 6.4.

Esperamos que essa primeira implementação da Semântica de Ações Orientada a Objetos encoraje o uso deste formalismo e também a continuidade do desenvolvimento da ferramenta descrita, além da implementação de outras ferramentas para o formalismo em linguagens orientada a objetos e com suporte a Visitor Patterns [GVJH98, AP03].

As principais contribuições deste trabalho podem ser sumarizadas da seguinte forma:

- Primeira implementação da Semântica de Ações Orientada a Objetos. Usando Maude foi possível desenvolver um ambiente executável para especificações de linguagens de programação usando um formalismo baseado em Semântica de Ações e orientação a objetos.

- MOOAT é uma extensão conservativa de Full Maude e MMT. Isso significa que além da Semântica de Ações Orientada a Objetos, Lógica de Reescrita e Semântica Operacional Estrutural Modular continuam sendo aceitas pelo ambiente.

- A MSOS da Semântica de Ações Orientada a Objetos foi definida. O uso da MMT possibilitou a implementação da Notação de Ações usando regras MSOS.

- A notação de MOOAT cobre um rico conjunto de ações e combinadores de ações, o qual inclui a notação completa da Semântica de Ações Orientada a Objetos original.

- A LFL foi reescrita. Foi proposta uma biblioteca mais limpa e usável por meio da sua própria reestruturação e pelo fato de trocar argumentos por *abstrações*.

- É possível usar o formalismo orientado a objetos de forma construtiva. Dessa forma, a Semântica de Ações Orientada a Objetos Construtiva foi proposta como uma alternativa a LFL.

- É possível usar tanto a LFL como a Semântica de Ações Orientada a Objetos Construtiva para descrever a semântica de linguagens independente de sintaxe. No caso da biblioteca, a hierarquia de classes fez isso possível. Já no caso da abordagem

construtiva, além da hierarquia os construtores também são responsáveis por tal característica.

- A questão de modularidade foi melhorada. Tanto a nova biblioteca como a nova abordagem construtiva melhoram significativamente os aspectos modulares presentes na abordagem orientada a objetos. Devido ao fato de que nas duas abordagens é possível escrever pedaços de especificações semânticas, os quais podem ser usados em vários projetos diferentes.

Como trabalhos futuros, visando o melhoramento do formalismo e da ferramenta, indicamos os seguintes itens:

- Implementação da faceta comunicativa na Notação de Ações de MOOAT.
- Implementação de mais funções auxiliares na Notação de Dados de MOOAT.
- Estudar e implementar formalmente a diretiva extends e a criação automática de objetos em MOOAT.
- Estudo da implementação da Semântica de Ações Orientada a Objetos em linguagens como C++ ou Java. Assim como investigar o uso de Visitor Patterns [MCM07] na geração de compiladores a partir de descrições semânticas neste formalismo.
- Estudar a possibilidade de especificar mais classes para a LFLv2, com o intuito de descrever linguagens de paradigmas diferentes para testar mais a fundo a nova versão da biblioteca.
- Investigar profundamente as vantagens da introdução dos conceitos da abordagem construtiva na abordagem orientada a objetos.
- Especificar linguagens mais complexas e de paradigmas diferentes para encontrar os limites de MOOAT e se possível corrigi-los.

BIBLIOGRAFIA

[AM04] Marcelo Araújo and Martin A. Musicante. Lfl: A library of generic classes for object-oriented action semantics. In *XXIV International Conference of the Chilean Computer Science Society (SCCC 2004), 11-12 November 2004, Arica, Chile*, pages 39–47. IEEE Computer Society, 2004.

[AP03] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2003.

[Ara04] Marcelo Araújo. Language features library - uma biblioteca de classes para semântica de ações orientada a objetos. Master's thesis, Universidade Federal do Paraná, 2004.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques and tools*. Addison-Wesley, 1988.

[BJM00] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1–2):35–132, 2000.

[Bro96] D. F. Brown. In *Sort Inference in Action Semantics - Phd. Thesis*. University of Glasgow, 1996.

[Car02] Cláudio Ricardo Vieira Carvilhe. Semântica de ações orientada a objetos. Master's thesis, Universidade Federal do Paraná, 2002.

[CB05] Fabricio Chalub and Christiano Braga. Maude msos tool. Technical report, Universidade Federal Fluminense, 2005. http://maude-msos-tool.sourceforge.net/mmt-manual.pdf.

[CB06] Fabricio Chalub and Christiano Braga. Maude msos tool. In Grit Denker and Carolyn Talcott, editors, *Proceedings of 6th International Workshop*

on Rewriting Logic and its Applications, WRLA, Vienna, Austria, April 2006. Elsevier.

[CDE+01] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[CDE+05] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Messeguer, and Carolyn Talcott. Maude manual (version 2.2), 2005. http://maude.cs.uiuc.edu/maude2-manual/html/.

[CM03] C. Carvilhe and M. A. Musicante. Object-oriented action semantics specifications. *Journal of Universal Computer Science*, 9(8):910–934, 2003.

[DHK96] Arie Van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping: An Algebraic Specification Approach: Vol. V.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.

[DM99] F. Durán and J. Meseguer. The Maude specification of Full Maude. Technical report, SRI International, 1999.

[DM01] K. Doh and P. Mosses. Composing programming languages by combining action semantics modules. In *First Workshop on Language Descriptions, Tools and Applications*, 2001.

[dOB01] Christiano de Oliveira Braga. *Rewriting Logic as a Semantics Framework for Modular Structural Operational Semantics.* PhD thesis, PUC-RIO, Dept of Informatics, 2001.

[dOBHMM00] Christiano de O. Braga, Edward Hermann Haeusler, José Meseguer, and Peter D. Mosses. Maude action tool: Using reflection to map action semantics to rewriting logic. In *AMAST '00: Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology*, pages 407–421, London, UK, 2000. Springer-Verlag.

[dR05] Fabricio Chalub Barbosa do Rosário. An implementation of modular structural operational semantics in maude. Master's thesis, Universidade Federal Fluminense, 2005.

[Dur99] Francisco Durán. *Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, University of Málaga, 1999.

[Gay02] J. E. Labra Gayo. Reusable semantic specifications of programming languages. In *SBLP 2002 - VI Simpósio Brasileiro de Linguagens de Programação*, 2002.

[GVJH98] Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. *Design Patterns CD: Elements of Reusable Object-Oriented Software, (CD-ROM)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[IM05] Jørgen Iversen and Peter D. Mosses. Constructive action semantics for core ML. *IEE Proceedings Software*, 2005. Special issue on Language Definitions and Tool Generation. To appear in Brics RS-04-37.

[Ive05] Jorgen Iversen. *Formalisms and tools supporting Constructive Action Semantics*. PhD thesis, BRICS International PhD School, May 2005.

[MB04] José Meseguer and Christiano Braga. Modular rewriting semantics of programming languages. In Charles Rattray, Savitri Maharaj, and Carron Shankland, editors, *In Algebraic Methodology and Software Technology: proceedings of the 10th International Conference, AMAST 2004*, volume 3116 of *LNCS*, pages 364–378, Stirling, Scotland, UK, July 2004. Springer. ISSN 0302-9743, ISBN 3-540-22381-9.

[MCM07] André Murbach Maidl, Cláudio Carvilhe, and Martin A. Musicante. Using visitor patterns in object-oriented action semantics. In *SBLP 2007 - XI Brazilian Symposium on Programming Languages*, 2007.

[MM94] Peter D. Mosses and Martin A. Musicante. An action semantics for ML concurrency primitives. Number 873 in Lecture Notes in Computer Science, Barcelona, Spain, October 1994. FME, Springer-Verlag.

[MM01] L. C. Menezes and H. Moura. Component-based action semantics: A new approach for programming language specifications. In *SBLP 2001 - V Simpósio Brasileiro de Linguagens de Programação*, pages 152–163, Paraná, 2001. Universidade Federal do Paraná.

[MOM93] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical report, SRI International, 1993.

[Mos92] P. Mosses. Action semantics. In *Action Semantics*. Cambridge University Press, 1992.

[Mos99] Peter D. Mosses. A modular SOS for Action Notation. Technical Report 99-56, 1999.

[Mos03] Peter D. Mosses. Modular SOS: Differences from SOS. Technical report, 2003.

[Mos04] Peter D. Mosses. Modular structural operational semantics. *J. Logic and Algebraic Programming*, 60–61:195–228, 2004. Special issue on SOS.

[Mos05] Peter D. Mosses. A constructive approach to language definition. *Journal of Universal Computer Science*, 11(7):1117–1134, July 2005.

[Mou93] H. Moura. In *Action Notation Transformations - Phd. Thesis*. University of Glasgow, 1993.

[Mus96] M. Musicante. In *On the Relational Semantics of Interleaving Constructors - Phd. Thesis*. Universidade Federal de Pernambuco, 1996.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing, 1992.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN-19, Aarhus University, Computer Science Department, 1981.

[Sch86] D. A. Schmidt. In *Denotational Semantics: A Methodology for Language Development*. Allyn Bacon, 1986.

[Seb00] Robert W. Sebesta. *Conceitos de Linguagens de Programação*. Bookman, 4a edição edition, 2000.

[vdBHdJ+01] Mark van den Brand, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter Olivier, Jeroen Scheerder, Jurgen Vinju, Eelco Visser, and Joost Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of Compiler Construction 2001 (CC 2001)*, LNCS. Springer, 2001.

[vdBIM06] Mark van den Brand, Jørgen Iversen, and Peter D. Mosses. An action environment. *Sci. Comput. Program.*, 61(3):245–264, 2006.

[VMO00] Alberto Verdejo and Narciso Martí-Oliet. Implementing CCS in Maude. In Tommaso Bolognesi and Diego Latella, editors, *Formal Methods For Distributed System Development. FORTE/PSTV 2000 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX) October 10–13, 2000, Pisa, Italy*, pages 351–366. Kluwer Academic Publishers, 2000.

[Wat91] D. Watt. In *Programming Language Syntax and Semantics*. Prentice Hall International (UK), 1991.

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

APÊNDICE A

CLASSES DA LFLV2

A.1 LFL.Declaration.Paradigm.Imper

```
Class Constant
    locating LFL.Declaration.Paradigm.Imper.Constant
    using I:Token, Y:Yielder
    semantics:
        elaborate-constant(I, Y) =
            | enact Y
            then
            | bind token I to the value
End Class

Class Variable
    locating LFL.Declaration.Paradigm.Imper.Variable
    using I:Token, T:Sort, Y:Yielder
    semantics:
        elaborate-variable(I, T, Y) =
            | enact Y and allocate a cell [ T ]
            then
            | | bind I to the given cell
            | and
            | | store the given value in the given cell
End Class

Class VariableDec
    locating LFL.Declaration.Paradigm.Imper.VariableDec
    using I:Token, T:Sort
    semantics:
        elaborate-variable(I, T) =
            | allocate a cell [ T ]
            then
            | bind token I to the given cell
End Class
```

A.2 LFL.Declaration.Shared

```
Class VariableSequence
    locating LFL.Declaration.Shared.VariableSequence
    using Y1:Yielder, Y2:Yielder
    semantics:
        elaborate-var-sequence(Y1,Y2) =
            | enact Y1 before enact Y2
End Class
```

A.3 LFL.Command.Paradigm.Imper

```
Class Assignment
    locating LFL.Command.Paradigm.Imper.Assignment
    using I:Token, Y:Yielder
    semantics:
        execute-assignment(I, Y) =
            | enact Y
            then
            | store the value in the cell bound to token I
End Class

Class Repeat
    locating LFL.Command.Paradigm.Imper.Repeat
    using Y1:Yielder, Y2:Yielder
    semantics:
        execute-repeat(Y1,Y2) =
            unfolding
            | enact Y1 and then enact Y2
            | | then unfold else complete
End Class
```

A.4 LFL.Command.Shared

```
Class Sequence
    locating LFL.Command.Shared.Sequence
    using Y1:Yielder, Y2:Yielder
    semantics:
        execute-sequence(Y1,Y2) =
            | enact Y1 and then enact Y2
End Class
```

Class Selection
    locating LFL.Command.Shared.Selection
    using Y_1:Yielder, Y_2:Yielder, Y_3:Yielder
    semantics:
        execute-if-then(Y_1,Y_2) =
            | enact Y_1 then enact Y_2 else complete
        execute-if-then-else(Y_1,Y_2,Y_3) =
            | enact A_1 then enact Y_2 else enact Y_3
End Class

Class While
    locating LFL.Command.Shared.While
    using Y_1:Yielder, Y_2:Yielder
    semantics:
        execute-while(Y_1,Y_2) =
            unfolding
            | enact Y_1 then
            | | enact Y_2 and then unfold else complete
End Class

A.5 LFL.Expression.Shared

Class Sum
    locating LFL.Expression.Shared.Sum
    using Y_1:Yielder, Y_2:Yielder
    semantics:
        evaluate-sum(Y_1,Y_2) =
            | enact Y_1 and enact Y_2
            then
            | give sum(the given integer #1, the given integer #2)

End Class

Class Subtract
    locating LFL.Expression.Shared.Subtract
    using Y_1:Yielder, Y_2:Yielder
    semantics:
        evaluate-sub(Y_1,Y_2) =
            | enact Y_1 and enact Y_2
            then
            | give difference(the given integer #1, the given integer #2)
End Class

```
Class Product
    locating LFL.Expression.Shared.Product
    using Y1:Yielder, Y2:Yielder
    semantics:
        evaluate-prod(Y1,Y2) =
            | enact Y1 and enact Y2
            then
            | give product(the given integer #1, the given integer #2)
End Class

Class Identifier
    locating LFL.Expression.Shared.Identifier
    using I:Token
    semantics:
        evaluate-identifier(I) =
            | give the value bound to I
            or
            | give the value stored in the cell bound to I
End Class

Class And
    locating LFL.Expression.Shared.And
    using Y1:Yielder, Y2:Yielder
    semantics:
        evaluate-and(Y1,Y2) =
            enact Y1 then
            | check (the given value) and then enact Y2
            or
            | check not (the given value) and then give false
End Class

Class Or
    locating LFL.Expression.Shared.Or
    using Y1:Yielder, Y2:Yielder
    semantics:
        evaluate-or(Y1,Y2) =
            enact Y1 then
            | check (the given value) and then give true
            or
            | check not (the given value) and then enact Y2
End Class

Class True
    locating LFL.Expression.Shared.True
    semantics:
        evaluate-true() =
            give true
End Class
```

```
Class False
    locating LFL.Expression.Shared.False
    semantics:
        evaluate-false() =
            give false
End Class

Class LessThan
    locating LFL.Expression.Shared.LessThan
    using Y1:Yielder, Y2:Yielder
    semantics:
        evaluate-less-than(Y1,Y2) =
            | enact Y1 and enact Y2
            then
            | give less(the given integer #1, the given integer #2)
End Class

Class Equality
    locating LFL.Expression.Shared.Equality
    using Y1:Yielder, Y2:Yielder
    semantics:
        evaluate-equality(Y1,Y2) =
            | enact Y1 and enact Y2
            then
            | give equal(the given integer #1, the given integer #2)
End Class
```

APÊNDICE B

SINTAXE DE MOOAT

(1) *ClassModule* ::= "class" *ClassName* "is"
⟨ *ClassExtends* ⟩* ⟨ *ClassDefinition* ⟩* "endclass"

(2) *ClassExtends* ::= "extends" *ClassName* ⟨ "," *ClassName* ⟩* "."

(3) *ClassDefinition* ::= ⟨ *SyntacticPart* ⟩* ⟨ *SemanticPart* ⟩*

(4) *SyntacticPart* ::= *SyntacticSort* "." | *SyntacticSort* "::=" *syntax-tree* "."

(5) *SemanticPart* ::= ⟨ *SemanticFunctions* ⟩* ⟨ *SemanticEquations* ⟩*

(6) *SemanticFunctions* ::= ⟨ *SemanticFunction* ⟩+

(7) *SemanticFunction* ::=
"absmethod" *FunctionName Tokens* "->" "Action" "." |
"absmethod" *FunctionName Tokens* "->" *Data* "."

(8) *Tokens* ::= *TokenName* ⟨ "," *Tokens* ⟩*

(9) *SemanticEquations* ::= ⟨ *SemanticEquation* ⟩+

(10) *SemanticEquation* ::=
"method" *FunctionName syntax-tree-with-objects* "=" *Action* "."
"method" *FunctionName syntax-tree-with-objects* "=" *Data* "."

(11) *ObjectDeclaration* ::= *Identifier* ":" *SyntacticSort*

(12) *Action* ::= *Action* "or" *Action* | "fail" | "commit" | *Action* "and" *Action* |
"complete" | "indivisibly" *Action* | *Action* "and then" *Action* |
Action "trap" *Action* | "escape" | "unfolding" *Action* | "unfold" |
"diverge" | "give" Yielder" | "regive" | "choose" *Yielder* |
"check" *Yielder* | *Action* "then" *Action* | "escape with" *Yielder* |
"bind" *Yielder* "to" *Yielder* | "rebind" | "unbind" *Yielder* |
"produce" *Yielder* | "furthermore" *Action* | *Action* "moreover" *Action* |
Action "hence" *Action* | *Action* "before" *Action* |
"store" *Yielder* "in" *Yielder* | "unstore" *Yielder* | "reserve" *Yielder* |
"unreserve" *Yielder* | "enact" *Yielder* |
"indirectly bind" *Yielder* "to" *Yielder* | "indirectly produce" *Yielder* |
"redirect" *Yielder* "to" *Yielder* | "undirect" *Yielder* |
"recursively bind" *Yielder* "to" *Yielder* |
Action "else" *Action* | "allocate" *Yielder*

(13) *Yielder* ::= *Data* | "a" *Yielder* | "the" *Yielder* | "nothing" |
"the" *DataSort* "yielded by" *Yielder* | "it" | "them" |
"given" *DataSort* | "given" *DataSort* # *Int* |
"current bindings" | "the" *DataSort* "bound to" *Yielder* |
Yielder "receiving" *Yielder* |
"current storage" | "the" *DataSort* "stored in" *Yielder* |
"application" *Yielder* "to" *Yielder* | "closure" *Yielder* |
"encapsulate" *Action* | "indirect closure" *Yielder*

(14) *Data* ::= *Datum* | *DataSort*

(15) *Datum* ::= "none" | "unknown" | "uninitialized" | *Abstraction* |
"<" *Int* ">" | "<" *Boolean* ">" | "<" *Token* ">" | "<" *Cell* ">" |
"<" *Transients* ">" | "<" *Bindings* ">" | "<" *Storage* ">" |

(16) *DataSort* ::= "integer" | "truth-value" | "token" | "cell" |
"abstraction" | "value"

APÊNDICE C

μ-PASCAL + OOAS + MOOAT

```
(class Identifier is
 Id .
 Id ::= @ Token .
 absmethod convert Id -> Token .
 method convert (@ T:Token) = T:Token .
endclass)

(class Declaration is
 Dec .
 absmethod elaborate Dec -> Action .
endclass)

(class Command is
 Cmd .
 absmethod execute Cmd -> Action .
endclass)

(class Expression is
 Exp .
 absmethod evaluate Exp -> Action .
endclass)

(class ConstantDec is
 extends Declaration .
 Dec ::= const Id = Exp .
 method elaborate (const I:Id = E:Exp) = (evaluate E:Exp)
    then (bind < convert I:Id > to given value) .
endclass)

(class VariableDec is
 extends Declaration .
 Dec ::= var Id .
 Dec ::= var Id = Exp .
 method elaborate (var I:Id) = (allocate a cell)
    then (bind < convert I:Id > to given cell) .
 method elaborate (var I:Id = E:Exp) =
    ((evaluate E:Exp) and (allocate a cell))
    then ((bind < convert I:Id > to given cell)
    and store given value in given cell) .
endclass)
```

```
(class DecSeq is
 extends Declaration .
 Dec ::= Dec ; Dec .
 method elaborate (D1:Dec ; D2:Dec) =
    elaborate D1:Dec before elaborate D2:Dec .
endclass)

(class Assign is
 extends Command .
 Cmd ::= Id := Exp .
 method execute (I:Id := E:Exp) = (evaluate E:Exp)
    then store given value in the cell bound to < convert I:Id > .
endclass)

(class Selection is
 extends Command .
 Cmd ::= if Exp then Cmd end-if .
 Cmd ::= if Exp then Cmd else Cmd end-if .
 method execute (if E:Exp then C1:Cmd end-if) =
    (evaluate E:Exp) then
    (execute C1:Cmd else complete) .
 method execute (if E:Exp then C1:Cmd else C2:Cmd end-if) =
    (evaluate E:Exp) then
    (execute C1:Cmd else execute C2:Cmd) .
endclass)

(class Repeat is
 extends Command .
 Cmd ::= repeat Cmd until Exp .
 method execute (repeat C:Cmd until E:Exp) = unfolding ((execute C:Cmd)
    and then ((evaluate E:Exp) then (unfold else complete))) .
endclass)

(class Sequence is
 extends Command .
 Cmd ::= Cmd ; Cmd .
 method execute (C1:Cmd ; C2:Cmd) =
    (execute C1:Cmd and then execute C2:Cmd) .
endclass)

(class Number is
 extends Expression .
 Exp ::= Int .
 method evaluate (N:Int) = give < N:Int > .
endclass)

(class IdExp is
```

```
 extends Expression .
 Exp ::= Id .
 method evaluate (I:Id) = give (the value bound to < convert I:Id >) or
    give (the value stored in the cell bound to < convert I:Id >) .
endclass)

(class Sum is
 extends Expression .
 Exp ::= Exp '+ Exp .
 method evaluate (E1:Exp '+ E2:Exp) =
    (evaluate E1:Exp and evaluate E2:Exp)
    then give sum(given integer # 1, given integer # 2) .
endclass)

(class Sub is
 extends Expression .
 Exp ::= Exp '- Exp .
 method evaluate (E1:Exp '- E2:Exp) =
    (evaluate E1:Exp and evaluate E2:Exp)
    then give difference(given integer # 1, given integer # 2) .
endclass)

(class Prod is
 extends Expression .
 Exp ::= Exp '* Exp .
 method evaluate (E1:Exp '* E2:Exp) =
    (evaluate E1:Exp and evaluate E2:Exp)
    then give product(given integer # 1, given integer # 2) .
endclass)

(class True is
 extends Expression .
 Exp ::= true .
 method evaluate (true) = give < tt > .
endclass)

(class False is
 extends Expression .
 Exp ::= false .
 method evaluate (false) = give < ff > .
endclass)

(class LessThan is
 extends Expression .
 Exp ::= Exp '< Exp .
 method evaluate (E1:Exp '< E2:Exp) =
    (evaluate E1:Exp and evaluate E2:Exp)
    then give less(given integer # 1, given integer # 2) .
```

```
endclass)

(class Equality is
 extends Expression .
 Exp ::= Exp '= Exp .
 method evaluate (E1:Exp '= E2:Exp) =
    (evaluate E1:Exp and evaluate E2:Exp)
    then give equal(given integer # 1, given integer # 2) .
endclass)

(class Micro-Pascal is
 Prog .
 Prog ::= begin Dec ; Cmd end .
 absmethod run Prog -> Action .
 method run (begin D:Dec ; C:Cmd end) =
    elaborate D:Dec hence execute C:Cmd .
endclass)

(class Factorial is
 extends ConstantDec, VariableDec, DecSeq .
 extends Assign, Selection, Repeat, Sequence .
 extends Number, IdExp, Sum, Sub, Prod .
 extends True, False, LessThan, Equality .
 extends Micro-Pascal .

 absmethod fat -> Action .
 method fat = (run
            (begin
                ((var @ 'a = 1) ;
                (var @ 'b = 0)) ;
                (repeat
                    ((@ 'a := (@ 'a '+ (@ 'a '* @ 'b))) ;
                    (@ 'b := (@ 'b '+ 1)))
                until (@ 'b '< 5))
            end)
            ) .
endclass)
```

APÊNDICE D

LFLV2 + MOOAT

```
(class LFL is
 extends STATE .
endclass)

(class LFL.Declaration is
 extends LFL .
endclass)

(class LFL.Declaration.Paradigm is
 extends LFL.Declaration .
endclass)

(class LFL.Declaration.Paradigm.Imper is
 extends LFL.Declaration.Paradigm .
endclass)

(class LFL.Declaration.Paradigm.Imper.Constant is
 extends LFL.Declaration.Paradigm.Imper .
 absmethod elaborate-constant (Token, Yielder) -> Action .
 method elaborate-constant (T:Token, Y:Yielder) =
    (enact Y:Yielder) then
    (bind < T:Token > to given value) .

endclass)

(class LFL.Declaration.Paradigm.Imper.Variable is
 extends LFL.Declaration.Paradigm.Imper .
 absmethod elaborate-variable (Token, Yielder) -> Action .
 method elaborate-variable (T:Token, Y:Yielder) =
    ((enact Y:Yielder) and (allocate a cell))
    then ((bind < T:Token > to given cell) and
    store given value in given cell) .
endclass)

(class LFL.Declaration.Paradigm.Imper.VariableDec is
 extends LFL.Declaration.Paradigm.Imper .
 absmethod elaborate-variable (Token) -> Action .
 method elaborate-variable (T:Token) =
    (allocate a cell) then
    (bind < T:Token > to given cell) .
```

```
endclass)

(class LFL.Declaration.Shared is
 extends LFL.Declaration .
endclass)

(class LFL.Declaration.Shared.DecSeq is
 extends LFL.Declaration.Shared .
 absmethod elaborate-decseq (Yielder, Yielder) -> Action .
 method elaborate-decseq (Y1:Yielder, Y2:Yielder) =
    enact Y1:Yielder before enact Y2:Yielder .
endclass)

(class LFL.Command is
 extends LFL .
endclass)

(class LFL.Command.Paradigm is
 extends LFL.Command .
endclass)

(class LFL.Command.Paradigm.Imper is
 extends LFL.Command.Paradigm .
endclass)

(class LFL.Command.Paradigm.Imper.Assignment is
 extends LFL.Command.Paradigm.Imper .
 absmethod execute-assignment (Token, Yielder) -> Action .
 method execute-assignment (T:Token, Y:Yielder) =
    (enact Y:Yielder) then
    (store given value in the cell bound to < T:Token >) .
endclass)

(class LFL.Command.Paradigm.Imper.Repeat is
 extends LFL.Command.Paradigm.Imper .
 absmethod execute-repeat (Yielder, Yielder) -> Action .
 method execute-repeat (Y1:Yielder, Y2:Yielder) = unfolding
    ((enact Y1:Yielder and then enact Y2:Yielder)
    then (unfold else complete)) .
endclass)

(class LFL.Command.Shared is
 extends LFL.Command .
endclass)

(class LFL.Command.Shared.Sequence is
 extends LFL.Command.Shared .
 absmethod execute-sequence (Yielder, Yielder) -> Action .
```

```
 method execute-sequence (Y1:Yielder, Y2:Yielder) =
    enact Y1:Yielder and then enact Y2:Yielder .
endclass)

(class LFL.Command.Shared.Selection is
 extends LFL.Command.Shared .
 absmethod execute-if-then (Yielder, Yielder) -> Action .
 absmethod execute-if-then-else (Yielder, Yielder, Yielder) -> Action .
 method execute-if-then (Y1:Yielder, Y2:Yielder) =
    (enact Y1:Yielder) then (enact Y2:Yielder else complete) .
 method execute-if-then-else (Y1:Yielder, Y2:Yielder, Y3:Yielder) =
    (enact Y1:Yielder) then (enact Y2:Yielder else enact Y3:Yielder) .
endclass)

(class LFL.Command.Shared.While is
 extends LFL.Command.Shared .
 absmethod execute-while (Yielder, Yielder) -> Action .
 method execute-while (Y1:Yielder, Y2:Yielder) = unfolding
    ((enact Y1:Yielder) then
    ((enact Y2:Yielder and then unfold) else (complete))) .
endclass)

(class LFL.Expression is
 extends LFL .
endclass)

(class LFL.Expression.Shared is
 extends LFL.Expression .
endclass)

(class LFL.Expression.Shared.Sum is
 extends LFL.Expression.Shared .
 absmethod evaluate-sum (Yielder, Yielder) -> Action .
 method evaluate-sum (Y1:Yielder, Y2:Yielder) =
    (enact Y1:Yielder and enact Y2:Yielder) then
    give sum(given integer # 1, given integer # 2) .
endclass)

(class LFL.Expression.Shared.Subtraction is
 extends LFL.Expression.Shared .
 absmethod evaluate-sub (Yielder, Yielder) -> Action .
 method evaluate-sub (Y1:Yielder, Y2:Yielder) =
    (enact Y1:Yielder and enact Y2:Yielder) then
    give difference(given integer # 1, given integer # 2) .
endclass)

(class LFL.Expression.Shared.Product is
 extends LFL.Expression.Shared .
```

```
 absmethod evaluate-prod (Yielder, Yielder) -> Action .
 method evaluate-prod (Y1:Yielder, Y2:Yielder) =
    (enact Y1:Yielder and enact Y2:Yielder) then
    give product(given integer # 1, given integer # 2) .
endclass)

(class LFL.Expression.Shared.Identifier is
 extends LFL.Expression.Shared .
 absmethod evaluate-identifier (Token) -> Action .
 method evaluate-identifier (T:Token) =
    give (the value bound to < T:Token >) or
    give (the value stored in the cell bound to < T:Token >) .
endclass)

(class LFL.Expression.Shared.And is
 extends LFL.Expression.Shared .
 absmethod evaluate-and (Yielder, Yielder) -> Action .
 method evaluate-and (Y1:Yielder, Y2:Yielder) =
    enact Y1:Yielder then
    ((check (given value) and then enact Y2:Yielder) or
    (check not (given value) and then give < ff >)) .
endclass)

(class LFL.Expression.Shared.Or is
 extends LFL.Expression.Shared .
 absmethod evaluate-or (Yielder, Yielder) -> Action .
 method evaluate-or (Y1:Yielder, Y2:Yielder) =
    enact Y1:Yielder then
    ((check (given value) and then give < tt >) or
    (check not (given value) and then enact Y2:Yielder)) .
endclass)

(class LFL.Expression.Shared.True is
 extends LFL.Expression.Shared .
 absmethod evaluate-true -> Action .
 method evaluate-true = give < tt > .
endclass)

(class LFL.Expression.Shared.False is
 extends LFL.Expression.Shared .
 absmethod evaluate-false -> Action .
 method evaluate-false = give < ff > .
endclass)

(class LFL.Expression.Shared.LessThan is
 extends LFL.Expression.Shared .
 absmethod evaluate-less (Yielder, Yielder) -> Action .
 method evaluate-less (Y1:Yielder, Y2:Yielder) =
```

```
    (enact Y1:Yielder and enact Y2:Yielder) then
    give less(given integer # 1, given integer # 2) .
endclass)

(class LFL.Expression.Shared.Equality is
 extends LFL.Expression.Shared .
 absmethod evaluate-equality (Yielder, Yielder) -> Action .
 method evaluate-equality (Y1:Yielder, Y2:Yielder) =
    (enact Y1:Yielder and enact Y2:Yielder) then
    give equal(given integer # 1, given integer # 2) .
endclass)
```

D.1 μ-Pascal + LFLv2 + MOOAT

```
(class Identifier is
 Id .
 Id ::= @ Token .
 absmethod convert Id -> Token .
 method convert (@ T:Token) = T:Token .
endclass)

(class Declaration is
 Dec .
 absmethod elaborate Dec -> Action .
endclass)

(class Command is
 Cmd .
 absmethod execute Cmd -> Action .
endclass)

(class Expression is
 Exp .
 absmethod evaluate Exp -> Action .
endclass)

(class ConstantDec is
 extends Declaration .
 extends LFL.Declaration.Paradigm.Imper.Constant .
 Dec ::= const Id = Exp .
 method elaborate (const I:Id = E:Exp) =
    elaborate-constant (convert I:Id, encapsulate (evaluate E:Exp)) .
endclass)

(class VariableDec is
 extends Declaration .
```

```
 extends LFL.Declaration.Paradigm.Imper.Variable .
 extends LFL.Declaration.Paradigm.Imper.VariableDec .
 Dec ::= var Id .
 Dec ::= var Id = Exp .
 method elaborate (var I:Id) =
    elaborate-variable (convert I:Id) .
 method elaborate (var I:Id = E:Exp) =
    elaborate-variable (convert I:Id, encapsulate (evaluate E:Exp)) .
endclass)

(class DecSeq is
 extends Declaration .
 extends LFL.Declaration.Shared.DecSeq .
 Dec ::= Dec ; Dec .
 method elaborate (D1:Dec ; D2:Dec) =
    elaborate-decseq (encapsulate (elaborate D1:Dec),
                      encapsulate (elaborate D2:Dec)) .
endclass)

(class Assign is
 extends Command .
 extends LFL.Command.Paradigm.Imper.Assignment .
 Cmd ::= Id := Exp .
 method execute (I:Id := E:Exp) =
    execute-assignment (convert I:Id, encapsulate (evaluate E:Exp)) .
endclass)

(class Selection is
 extends Command .
 extends LFL.Command.Shared.Selection .
 Cmd ::= if Exp then Cmd end-if .
 Cmd ::= if Exp then Cmd else Cmd end-if .
 method execute (if E:Exp then C1:Cmd end-if) =
    execute-if-then (encapsulate (evaluate E:Exp),
                     encapsulate (execute C1:Cmd)) .
 method execute (if E:Exp then C1:Cmd else C2:Cmd end-if) =
    execute-if-then-else (encapsulate (evaluate E:Exp),
    encapsulate (execute C1:Cmd), encapsulate (execute C2:Cmd)) .
endclass)

(class Repeat is
 extends Command .
 extends LFL.Command.Paradigm.Imper.Repeat .
 Cmd ::= repeat Cmd until Exp .
 method execute (repeat C:Cmd until E:Exp) =
    execute-repeat (encapsulate (execute C:Cmd),
                    encapsulate (evaluate E:Exp)) .
endclass)
```

```
(class Sequence is
 extends Command .
 extends LFL.Command.Shared.Sequence .
 Cmd ::= Cmd ; Cmd .
 method execute (C1:Cmd ; C2:Cmd) =
    execute-sequence (encapsulate (execute C1:Cmd),
                      encapsulate (execute C2:Cmd)) .
endclass)

(class Number is
 extends Expression .
 Exp ::= Int .
 method evaluate (N:Int) = give < N:Int > .
endclass)

(class IdExp is
 extends Expression .
 extends LFL.Expression.Shared.Identifier .
 Exp ::= Id .
 method evaluate (I:Id) =
    evaluate-identifier (convert I:Id) .
endclass)

(class Sum is
 extends Expression .
 extends LFL.Expression.Shared.Sum .
 Exp ::= Exp '+ Exp .
 method evaluate (E1:Exp '+ E2:Exp) =
    evaluate-sum (encapsulate (evaluate E1:Exp),
                  encapsulate (evaluate E2:Exp)) .
endclass)

(class Sub is
 extends Expression .
 extends LFL.Expression.Shared.Subtraction .
 Exp ::= Exp '- Exp .
 method evaluate (E1:Exp '- E2:Exp) =
    evaluate-sub (encapsulate (evaluate E1:Exp),
                  encapsulate (evaluate E2:Exp)) .
endclass)

(class Prod is
 extends Expression .
 extends LFL.Expression.Shared.Product .
 Exp ::= Exp '* Exp .
 method evaluate (E1:Exp '* E2:Exp) =
    evaluate-prod (encapsulate (evaluate E1:Exp),
```

```
                  encapsulate (evaluate E2:Exp)) .
endclass)

(class True is
 extends Expression .
 extends LFL.Expression.Shared.True .
 Exp ::= true .
 method evaluate (true) =
    evaluate-true .
endclass)

(class False is
 extends Expression .
 extends LFL.Expression.Shared.False .
 Exp ::= false .
 method evaluate (false) =
    evaluate-false .
endclass)

(class LessThan is
 extends Expression .
 extends LFL.Expression.Shared.LessThan .
 Exp ::= Exp '< Exp .
 method evaluate (E1:Exp '< E2:Exp) =
    evaluate-less (encapsulate (evaluate E1:Exp),
                   encapsulate (evaluate E2:Exp)) .
endclass)

(class Equality is
 extends Expression .
 extends LFL.Expression.Shared.Equality .
 Exp ::= Exp '= Exp .
 method evaluate (E1:Exp '= E2:Exp) =
    evaluate-equality (encapsulate (evaluate E1:Exp),
                       encapsulate (evaluate E2:Exp)) .
endclass)

(class Micro-Pascal is
 Prog .
 Prog ::= begin Dec ; Cmd end .
 absmethod run Prog -> Action .
 method run (begin D:Dec ; C:Cmd end) =
    elaborate D:Dec hence execute C:Cmd .
endclass)

(class Factorial is
 extends ConstantDec, VariableDec, DecSeq .
 extends Assign, Selection, Repeat, Sequence .
```

```
 extends Number, IdExp, Sum, Sub, Prod .
 extends True, False, LessThan, Equality .
 extends Micro-Pascal .

 absmethod fat -> Action .
 method fat = (run
            (begin
                ((var @ 'a = 1) ;
                (var @ 'b = 0)) ;
                (repeat
                    ((@ 'a := (@ 'a '+ (@ 'a '* @ 'b))) ;
                    (@ 'b := (@ 'b '+ 1)))
                until (@ 'b '< 5))
            end)
          ) .
endclass)
```

APÊNDICE E

COOAS + MOOAT

```
(class Exp is
 Exp .
 absmethod evaluate Exp -> Action .
endclass)

(class Exp/Val is
 extends Exp .
 Value .
 Value ::= Int .
 Value ::= Boolean .
 Exp ::= Value .
 method evaluate (V:Value) = give < V:Value > .
endclass)

(class Exp/Val-Id is
 extends Exp .
 Exp ::= val (Token) .
 method evaluate (val(T:Token)) =
    give (the value bound to < T:Token >) or
    give (the value stored in the cell bound to < T:Token >) .
endclass)

(class Exp/Sum is
 extends Exp .
 Exp ::= exp-sum (Exp, Exp) .
 method evaluate (exp-sum(E1:Exp, E2:Exp)) =
    (evaluate E1:Exp and evaluate E2:Exp) then
    give sum(given integer # 1, given integer # 2) .
endclass)

(class Exp/Sub is
 extends Exp .
 Exp ::= exp-sub (Exp, Exp) .
 method evaluate (exp-sub(E1:Exp, E2:Exp)) =
    (evaluate E1:Exp and evaluate E2:Exp) then
    give difference(given integer # 1, given integer # 2) .
endclass)

(class Exp/Prod is
 extends Exp .
```

```
 Exp ::= exp-prod (Exp, Exp) .
 method evaluate (exp-prod(E1:Exp, E2:Exp)) =
    (evaluate E1:Exp and evaluate E2:Exp) then
    give product(given integer # 1, given integer # 2) .
endclass)

(class Exp/And is
 extends Exp .
 Exp ::= exp-and (Exp, Exp) .
 method evaluate (exp-and(E1:Exp, E2:Exp)) =
    evaluate E1:Exp then
    ((check (given value) and then evaluate E2:Exp) or
    (check not (given value) and then give < ff >)) .
endclass)

(class Exp/Or is
 extends Exp .
 Exp ::= exp-or (Exp, Exp) .
 method evaluate (exp-or(E1i:Exp, E2:Exp)) =
    evaluate E1:Exp then
    ((check (given value) and then give < tt >) or
    (check not (given value) and then evaluate E2:Exp)) .
endclass)

(class Exp/True is
 extends Exp .
 Exp ::= truev .
 method evaluate (truev) = give < tt > .
endclass)

(class Exp/False is
 extends Exp .
 Exp ::= falsev .
 method evaluate (falsev) = give < ff > .
endclass)

(class Exp/LessThan is
 extends Exp .
 Exp ::= exp-less (Exp, Exp) .
 method evaluate (exp-less(E1:Exp, E2:Exp)) =
    (evaluate E1:Exp and evaluate E2:Exp) then
    give less(given integer # 1, given integer # 2) .
endclass)

(class Exp/Equality is
 extends Exp .
 Exp ::= exp-equal (Exp, Exp) .
 method evaluate (exp-equal(E1:Exp, E2:Exp)) =
```

```
    (evaluate E1:Exp and evaluate E2:Exp) then
    give equal(given integer # 1, given integer # 2) .
endclass)

(class Cmd is
 Cmd .
 absmethod execute Cmd -> Action .
endclass)

(class Cmd/Assignment is
 extends Cmd .
 Cmd ::= cmd-assign (Token, Exp) .
 method execute (cmd-assign(T:Token, E:Exp)) =
    (evaluate E:Exp) then
    (store given integer in the cell bound to < T:Token >) .
endclass)

(class Cmd/Repeat is
 extends Cmd .
 Cmd ::= cmd-repeat (Cmd, Exp) .
 method execute (cmd-repeat(C:Cmd, E:Exp)) =
    unfolding ((execute C:Cmd and then evaluate E:Exp)
    then (unfold else complete)) .
endclass)

(class Cmd/Sequence is
 extends Cmd .
 Cmd ::= cmd-seq (Cmd, Cmd) .
 method execute (cmd-seq(C1:Cmd, C2:Cmd)) =
    execute C1:Cmd and then execute C2:Cmd .
endclass)

(class Cmd/Cond is
 extends Cmd .
 Cmd ::= cmd-cond1 (Exp, Cmd) .
 Cmd ::= cmd-cond2 (Exp, Cmd, Cmd) .
 method execute (cmd-cond1(E:Exp, C1:Cmd)) =
    (evaluate E:Exp) then
    (execute C1:Cmd else complete) .
 method execute (cmd-cond2(E:Exp, C1:Cmd, C2:Cmd)) =
    (evaluate E:Exp) then
    (execute C1:Cmd else execute C2:Cmd) .
endclass)

(class Cmd/While is
 extends Cmd .
 Cmd ::= cmd-while (Exp, Cmd) .
 method execute (cmd-while(E:Exp, C:Cmd)) = unfolding
```

```
    (evaluate E:Exp then
    ((execute C:Cmd and then unfold) else complete)) .
endclass)

(class Dec is
 Dec .
 absmethod elaborate Dec -> Action .
endclass)

(class Dec/Constant is
 extends Dec .
 Dec ::= dec-const (Token, Exp) .
 method elaborate (dec-const(T:Token, E:Exp)) =
    evaluate E:Exp then
    bind < T:Token > to given value .
endclass)

(class Dec/Variable is
 extends Dec .
 Dec ::= dec-var (Token, Exp) .
 method elaborate (dec-var(T:Token, E:Exp)) =
    (evaluate E:Exp and allocate a cell)
    then ((bind < T:Token > to given cell)
    and (store given value in given cell)) .
endclass)

(class Dec/VarDec is
 extends Dec .
 Dec ::= dec-variable (Token) .
 method elaborate (dec-variable(T:Token)) =
    (allocate a cell) then
    (bind < T:Token > to given cell) .
endclass)

(class Dec/DecSeq is
 extends Dec .
 Dec ::= dec-seq (Dec, Dec) .
 method elaborate (dec-seq(D1:Dec, D2:Dec)) =
    elaborate D1:Dec before elaborate D2:Dec .
endclass)

(class Prog is
 Prog .
 Prog ::= prog (Dec, Cmd) .
 absmethod run Prog -> Action .
 method run (prog (D:Dec, C:Cmd)) =
    elaborate D:Dec hence execute C:Cmd .
endclass)
```

E.1 μ-Pascal + COOAS + MOOAT

```
(class Micro-Pascal is
 extends Exp .
 extends Exp/Val .
 extends Exp/Val-Id .
 extends Exp/Sum .
 extends Exp/Sub .
 extends Exp/Prod .
 extends Exp/True .
 extends Exp/False .
 extends Exp/LessThan .
 extends Exp/Equality .

 extends Cmd .
 extends Cmd/Assignment .
 extends Cmd/Repeat .
 extends Cmd/Sequence .
 extends Cmd/Cond .

 extends Dec .
 extends Dec/Variable .
 extends Dec/DecSeq .

 extends Prog .
endclass)

(class Factorial is
 extends Micro-Pascal .

 absmethod fat -> Action .
 method fat = (run
                (prog(
                    dec-seq(dec-var('a, 1),dec-var('b, 0)),
                    cmd-repeat(cmd-seq(
                    cmd-assign('a,exp-sum(val('a),
                               exp-prod(val('a),val('b)))),
                    cmd-assign('b,exp-sum(val('b), 1))),
                    exp-less(val('b), 5)))
                    )
                ) .
endclass)
```