

CLÁUDIO RICARDO VIEIRA CARVILLE

SEMÂNTICA DE AÇÕES ORIENTADA A OBJETOS

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

Orientador: Prof. Martin Musicante

CURITIBA

2002



Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

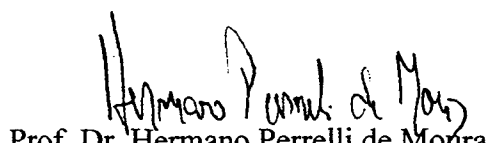
PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática do aluno *Cláudio Ricardo Vieira Carvilhe*, avaliamos o trabalho intitulado, "*Semântica de Ações Orientada a Objetos*", cuja defesa foi realizada no dia 15 de julho de 2002, às dez horas, no anfiteatro A do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

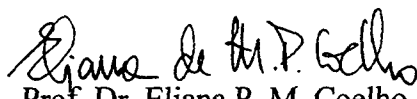
Curitiba, 15 de julho de 2002.



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



Prof. Dr. Hermano Perrelli de Moura
CIN/UFPE - Membro Externo



Prof. Dr. Eliana P. M. Coelho
DINF/UFPR

AGRADECIMENTOS

À Cintia Machado pela motivação e apoio irrestrito em todas as fases deste projeto.

À minha família pelo suporte fornecido em todos estes anos.

Ao professor Dr. Martin A. Musicante pela motivação e constante interesse no meu trabalho.

Aos amigos Fernanda Hembecker e Mauro A. Borchardt pelo companheirismo nos intervalos deste trabalho.

Aos professores Dr. Edson E. Scalabrin, Dra. Tamia M. Yamamoto e Dr. Robert C. Burnett pela colaboração junto à PUCPR para a divulgação deste trabalho em eventos científicos.

Aos professores MSc. Orlando A. Soares, Dr. Alcides Calsavara, Dr. Alceu S. B. Júnior, Dr. Marcos A. H. Shmeil, Dr. Hermano P. Moura e Dra. Eliana M. P. Coelho, pelos comentários e importantes sugestões na elaboração deste documento.

À Universidade Federal do Paraná (UFPR) pela oportunidade concedida.

À Pontifícia Universidade Católica do Paraná (PUCPR) pela ajuda financeira para a divulgação deste trabalho.

SUMÁRIO

AGRADECIMENTOS	i
RESUMO	vii
ABSTRACT	viii
1 INTRODUÇÃO	1
1.1 Definição do Problema	2
1.2 Objetivos	3
1.3 Estrutura desta Dissertação	3
2 SEMÂNTICA DE LINGUAGENS DE PROGRAMAÇÃO	5
2.1 Sintaxe de Linguagens de Programação	5
2.1.1 Sintaxe Abstrata	5
2.1.2 Sintaxe Concreta	6
2.2 Semântica Denotacional	7
2.2.1 Funções Semânticas e Equações Semânticas	7
2.2.2 O Modelo de Armazenamento - <i>Storage</i>	8
2.2.3 O Modelo de Ambientes - <i>Environment</i>	10
2.3 Semântica Operacional	11
2.3.1 Interpretadores de definição	11
2.3.2 Semântica de Transição	13
2.3.2.1 Uma linguagem imperativa simples	14
2.3.2.2 Semântica de passo misto	17
2.3.3 Semântica Relacional	18
2.3.3.1 Uma linguagem imperativa simples	19
3 SEMÂNTICA DE AÇÕES	22
3.1 Introdução	22

3.2	Notação de Dados	22
3.3	Notação de Ações	23
3.3.1	Faceta Básica	24
3.3.2	Faceta Funcional	25
3.3.3	Faceta Imperativa	26
3.3.4	Faceta Declarativa	27
3.3.5	Exemplo da linguagem imperativa simples	28
3.3.5.1	Sintaxe abstrata	28
3.3.5.2	Entidades semânticas	29
3.3.5.3	Funções semânticas	29
4	SEMÂNTICA DE AÇÕES BASEADA EM MÓDULOS E COMPONENTES	32
4.1	Semântica de Ações baseada em Módulos	32
4.1.1	Notação e exemplo do uso de módulos	32
4.2	Semântica de Ações baseada em Componentes	34
4.2.1	Notação de Componentes e Linguagens de Programação	34
4.3	Aspectos importantes no uso de Módulos e Componentes	36
5	SEMÂNTICA DE AÇÕES ORIENTADA A OBJETOS	38
5.1	Introdução	38
5.1.1	Classes e Objetos	39
5.1.1.1	Atributos e Operações dos Objetos	39
5.1.1.2	Generalização e Especialização	39
5.1.2	Semântica de Ações Orientada a Objetos	40
5.1.3	Implicações iniciais no tratamento de Módulos como Objetos	42
5.2	Sintaxe da Notação	42
5.2.1	Classes	43
5.2.1.1	Definição das Classes	44
5.2.1.2	Notação de Ações	46
5.3	Semântica da Notação	46
5.3.1	A Classe <i>State</i>	47
5.3.2	Atributos da classe <i>State</i>	47
5.3.2.1	Operações sobre a classe <i>State</i>	48
5.3.3	Operações Básicas	48

5.3.4	Operações Funcionais	52
5.3.5	Operações Declarativas	54
5.3.6	Operações Imperativas	58
5.3.7	Operações Reflexivas	59
5.3.8	Operações Híbridas	61
5.3.9	Sub-classes de <i>State</i>	62
6	ESTUDO DE CASO: A LINGUAGEM PURPLE	64
6.1	Sintaxe da Linguagem PURPLE	64
6.2	Um programa para cálculo do fatorial na Linguagem PURPLE	67
6.3	Semântica de Ações Orientada a Objetos da Linguagem PURPLE	67
6.3.1	Entidades Semânticas	68
6.3.2	Classes	68
6.3.2.1	Identificadores, Numerais e Tipos	68
6.3.2.2	Declarações	69
6.3.2.3	Comandos	71
6.3.2.4	Expressões	75
6.3.2.5	A Linguagem	79
6.4	Estendendo a especificação - A Linguagem <i>X-Purple</i>	79
6.4.1	Sintaxe da Linguagem <i>X-Purple</i>	80
6.4.2	Declaração de Funções e Procedimentos	80
6.4.3	Parâmetros Formais e Parâmetros Reais	82
6.4.3.1	Chamadas a procedimentos e Aplicação de funções	83
6.4.4	A Linguagem Estendida	84
7	CONCLUSÕES E TRABALHOS FUTUROS	85
APÊNDICES		
A	SEMÂNTICA DE AÇÕES ORIENTADA A OBJETOS	91
A.1	Sintaxe da Notação	91
A.2	Semântica da Notação	92
A.2.1	Comportamento das operações sobre a Classe <i>State</i>	92
A.2.2	Operações básicas	92
A.2.3	Operações funcionais	93
A.2.4	Operações declarativas	94

A.2.5	Operações imperativas	95
A.2.6	Operações reflexivas	95
A.2.7	Operações híbridas	96
A.2.8	Ambiente para definição de métodos	96
B	SEMÂNTICA DE AÇÕES ORIENTADA A OBJETOS DA LINGUAGEM PURPLE	97
B.1	Sintaxe Abstrata da Linguagem PURPLE	97
B.2	Semântica da Linguagem PURPLE	98
B.2.1	Entidades Semânticas	98
B.2.2	Identificadores, Numerais e Tipos	98
B.2.3	Declarações	99
B.2.4	Comandos	100
B.2.5	Expressões	102
B.2.6	Linguagem	104

LISTA DE FIGURAS

2.1	Modelo de armazenamento.	9
5.1	Estrutura de objetos da linguagem do Exemplo 5.1.1	40
6.1	Hierarquia de classes de Declarações na Linguagem PURPLE	70
6.2	Hierarquia de classes de Comandos na Linguagem PURPLE	72
6.3	Hierarquia de classes de Expressões na Linguagem PURPLE	75
6.4	Hierarquia de classes de Declarações na Linguagem <i>X-Purple</i>	81
6.5	Hierarquia de classes de Comandos na Linguagem <i>X-Purple</i>	83
6.6	Hierarquia de classes de Expressões na Linguagem <i>X-Purple</i>	83

RESUMO

Neste trabalho apresentamos a Semântica de Ações Orientada a Objetos, uma nova forma de tratar alguns dos problemas relacionados ao reuso e extensão de especificações de linguagens de programação em Semântica de Ações. A partir da análise de métodos para a divisão da especificação formal em partes, propomos uma forma de organizar as especificações, baseada em uma hierarquia de classes de objetos. Os objetos, que determinam a semântica das frases da linguagem, podem ser reutilizados e adequados a novas especificações de linguagens. Uma nova notação é definida e sua semântica é apresentada. Finalmente, um estudo de caso é elaborado, considerando a especificação formal de uma linguagem imperativa simples. Este estudo nos permite avaliar a abordagem proposta.

ABSTRACT

In this paper we present Object-Oriented Action Semantics, as a means for handling some problems related to reuse and extension of Action Semantics specifications. Based on the analysis of formal specification parts division methods, we propose a way to organize specifications based on a class hierarchy. Objects that define languages sentences semantics can be reused and applied in new language specifications. A new notation is defined and its semantics is presented. Finally, a case study is shown by considering a small imperative language formal specification. This case study allows us to assess our proposal.

CAPÍTULO 1

INTRODUÇÃO

Definimos a Semântica Formal de uma linguagem de programação como a especificação de um modelo matemático que a represente. Esta abordagem permite definir as estruturas da linguagem e prever o comportamento de seus programas. As especificações no paradigma de Semântica Formal caracterizam-se pela representação do *significado* dos programas a partir da sintaxe ou estrutura da linguagem. Neste documento introduzimos algumas abordagens para este fim: Semântica Denotacional [Wat91, Sch86], Semântica Operacional [Plo81, Win93] e Semântica de Ações [Mos92].

Semântica Denotacional constitui uma técnica pioneira para especificação de linguagens de programação. Inicialmente denominada “semântica matemática”, esta foi criada por Christopher Strachey e Dana Scott [SS71] e utiliza conceitos matemáticos bastante abstratos [Win93]. Esta técnica permite grande detalhamento da especificação pela riqueza da teoria matemática. Todavia, as especificações resultantes são geralmente complexas, o que dificulta sua compreensão, modificação e extensão.

Semântica de Ações [Mos92] é um formalismo onde o significado de um programa é expresso por uma entidade matemática denominada *ação*. Baseada em Semântica Denotacional, a Semântica de Ações adota a mesma estrutura de especificação dividida em: sintaxe abstrata, onde definimos a estrutura das frases da linguagem (declarações, comandos, expressões); e semântica, onde representamos o significado das frases.

Em Semântica de Ações representamos a semântica de um programa mediante a utilização de um conjunto padrão de ações (Action Notation), representadas por palavras da língua inglesa, em substituição à notação funcional da semântica denotacional. Isto tem contribuído para a criação de especificações mais compreensíveis, visando maior aceitação das descrições em semântica formal por parte da comunidade.

1.1 Definição do Problema

A proposta de uma nova linguagem de programação baseia-se em conceitos e estruturas presentes em linguagens já existentes, tais como: declaração de variáveis, definição de comandos e expressões. Desta forma, verificamos um requerimento importante às descrições em Semântica de Ações: o reaproveitamento de partes de especificação de linguagens na criação de novos projetos. Isto implica em prover mecanismos que permitam a divisão da especificação em partes que possam ser reutilizadas e também estendidas.

Inicialmente, foi proposto em [DM01] um método para criação de *módulos* em semântica de ações. Partes da especificação de uma linguagem de programação podem ser definidas separadamente, o que permite: a visão isolada dos elementos de especificação; a composição dos módulos (inserção de módulos no momento da definição de outros); e a reutilização das partes de especificação em outros projetos. Em seguida, em [MM01] foi proposta uma forma de expandir ainda mais a funcionalidade dos módulos, criando os denominados *componentes*, o que possibilita: a definição de estruturas básicas (componentes genéricos) que reduzem consideravelmente o tamanho das especificações; a criação de uma biblioteca de componentes; e a definição de linguagens de programação pelo uso de notação específica.

A reutilização e extensão de partes de especificação com o uso de módulos mostrou sucesso nos trabalhos de [DM01] e [MM01]. Todavia, verificamos algumas limitações no uso de módulos ou componentes, as quais detalharemos no Capítulo 4.

Uma dificuldade no esquema introduzido por [DM01] está no controle do acoplamento entre módulos. Neste trabalho, a divisão da especificação em partes é efetuada com a necessidade de inter-relacionamento das mesmas. A questão é que os quesitos de re-utilização e extensão, motivadores da proposta, só podem ser efetuados se um módulo faz referência a outro. Este problema de interdependência, torna a especificação propensa a inconsistências, relatadas em [DM01].

Em nossa visão, uma possibilidade de solução deste problema está no tratamento dos módulos como objetos, inspirado na abordagem de Orientação a Objetos [Rum94, Mey97]. A estruturação de interfaces bem definidas, e a criação de uma hierarquia de classes podem ser úteis na redução do acoplamento entre partes da especificação.

1.2 Objetivos

Nosso trabalho tem como objetivo principal a criação de uma nova abordagem em Semântica de Ações, intitulada Semântica de Ações Orientada a Objetos. Baseados nos trabalhos de [DM01] e [MM01], pretendemos prover o reuso e a extensão de descrições em semântica de ações, pela utilização de objetos.

A definição desta nova abordagem requer a especificação de uma notação que possibilite a definição dos diversos conceitos de orientação a objetos, tendo como foco principal a criação de classes. A sintaxe da notação, para fins de padronização, está baseada na sintaxe estabelecida por [DM01] para os módulos, agregando algumas estruturas adicionais. Um requisito básico deste trabalho é introduzir as noções de orientação a objetos sem afetar a notação padrão de ações [Mos92]. A finalidade desta condição é facilitar a adequação de especificações existentes em semântica de ações à nossa abordagem.

A semântica desta notação também é definida neste trabalho. A mesma baseia-se na utilização de uma classe principal - denominada *State* - que encapsula diversas funcionalidades da semântica de ações. As ações, por sua vez, são vistas como métodos sobre esta classe principal, os quais modificam a informação corrente.

Todas as classes, utilizando nossa abordagem, são por definição sub-classes de *State*, e ao mesmo tempo herdam todas as suas funcionalidades. A semântica dos métodos (ações) será expressa por meio de regras de inferência, com base em [Mou93].

Com o intuito de validar nossa notação, vamos estabelecer um estudo de caso que consiste na especificação em Semântica de Ações Orientada a Objetos de uma linguagem imperativa simples. Esta linguagem - denominada PURPLE - contém: declarações de variáveis e constantes, comandos e expressões. Baseados nesta linguagem vamos construir uma hierarquia de objetos que expressem sua semântica.

Além disso, propomos uma extensão desta linguagem, visando adicionar procedimentos e funções à estrutura da linguagem original. Com esta nova linguagem, intitulada *X-Purple*, analisamos a repercussão da inserção destas novas estruturas e os procedimentos para a criação da nova especificação.

1.3 Estrutura desta Dissertação

Este trabalho é composto por 7 capítulos. No Capítulo 2 vamos introduzir as noções necessárias ao desenvolvimento de nosso trabalho. O Capítulo 3 está destinado à descrição de Semântica de Ações, o formalismo base para nossa proposta. No Capítulo 4 apresentaremos a utilização de

módulos e componentes em Semântica de Ações. Semântica de Ações Orientada a Objetos será apresentada no Capítulo 5, sendo que a sintaxe e a semântica da notação serão apresentadas com a utilização de exemplos. O Capítulo 6 define o estudo de caso através da especificação formal da linguagem PURPLE e de sua extensão (*X-Purple*). Nossas conclusões e planos futuros estão indicadas no Capítulo 7.

CAPÍTULO 2

SEMÂNTICA DE LINGUAGENS DE PROGRAMAÇÃO

Expressar o significado de uma linguagem de programação é especificar o comportamento de seus programas. O funcionamento de um programa pode ser definido pela descrição de suas partes: comandos, declarações, expressões, etc. Neste Capítulo vamos apresentar algumas abordagens para expressar a Semântica de Linguagens de Programação: Semântica Denotacional e Semântica Operacional.

2.1 Sintaxe de Linguagens de Programação

A sintaxe de uma linguagem corresponde ao conjunto de regras para a escrita de seus programas. Estas regras nos fornecem informação acerca da estrutura da linguagem, o que nos permite determinar em que casos os programas estão no formato correto. O grau de detalhamento da sintaxe de uma linguagem varia de acordo com o que desejamos expressar. Vamos diferenciar aqui duas formas de apresentar a sintaxe de uma linguagem: sintaxe abstrata e sintaxe concreta [ASU88]. Em ambos os casos utilizamos a notação de Backus-Naur (BNF) [ASU88] para especificar a sintaxe.

2.1.1 Sintaxe Abstrata

Em sintaxe abstrata definimos uma linguagem em sua forma mais simples. Nesta abordagem, não há preocupação quanto a redundância e ambiguidade - definimos a linguagem simplesmente com o intuito de expressar sua estrutura em uma descrição de alto nível. Portanto, podemos omitir algumas definições, evidenciando assim apenas o que é essencial para expressar a linguagem e posteriormente sua semântica. Vamos nos basear no seguinte exemplo de linguagem:

Exemplo 2.1.1 (Sintaxe Abstrata de uma linguagem de expressões aritméticas)

$$e ::= e + e \mid e - e \mid e * e \mid e / e \mid n \quad (2.1)$$

Na equação (2.1) definimos indutivamente a sintaxe de uma linguagem de expressões aritméticas. O elemento sintático n representa um valor real (base da indução). O elemento sintático e aparece repetidas vezes na especificação, representando elementos do conjunto que está sendo definido (expressões aritméticas). As informações relevantes à estrutura da linguagem incluídas nesta abordagem que devemos considerar são apenas: suas sub-expressões e seus operadores.

2.1.2 Sintaxe Concreta

Na descrição da sintaxe concreta de uma linguagem são definidos tanto os elementos da linguagem quanto a forma em que estes são obtidos.

Para exemplificar esta abordagem vamos reescrever a estrutura da linguagem apresentada no exemplo (2.1.1). Definimos a sintaxe concreta daquela linguagem mediante o exemplo a seguir:

Exemplo 2.1.2 (Sintaxe Concreta de uma linguagem de expressões aritméticas)

$$e ::= e + t \mid e - t \mid t \quad (2.2)$$

$$t ::= t * f \mid t / f \mid f \quad (2.3)$$

$$f ::= n \quad (2.4)$$

Nesta nova estrutura, utilizamos os símbolos não-terminais e , t e f em complemento ao que definimos na sintaxe abstrata da linguagem. O símbolo e corresponde ao símbolo inicial (expressões), enquanto que os símbolos t e f representam os *termos* e *fatores*, respectivamente. As leis de formação (regras de produção) das expressões estão descritas como segue.

Em (2.2) definimos que uma expressão e pode ser uma soma ($e + t$), uma subtração de expressões ($e - t$) ou pode ser um termo (t). Um termo (2.3), por sua vez, pode ser uma multiplicação de um termo por um fator ($t * f$), uma divisão dos mesmos elementos (t / f) ou simplesmente um fator (f). Um fator, conforme (2.4), é um número (n).

A intenção nesta subdivisão da expressão em termos e fatores é simplesmente garantir uma possibilidade única de *parsing*, evitando a construção ambígua. Os fatores são mais restritos que os termos. Então, em uma expressão da forma “ $3 + 5 * 8$ ”, os elementos que compõem o termo “ $5 * 8$ ” vão estar agrupados juntos, uma vez que “ $3 + 5$ ” não é um fator.

Portanto, esta abordagem é interessante para a criação de um *interpretador* ou *compilador* da linguagem, sendo descartada quando o objetivo é mostrar o significado da mesma.

2.2 Semântica Denotacional

Um formalismo muito difundido para a descrição semântica de linguagens de programação é o da Semântica Denotacional [Wat91] [Sch86]. Inicialmente denominada *Semântica Matemática*, a Semântica Denotacional foi desenvolvida por Christopher Strachey baseando-se em fundamentos matemáticos fornecidos por Dana Scott [SS71]. Neste paradigma, a semântica de qualquer parte de um programa - suas declarações, comandos, expressões, etc. - é dita sua *denotação*. Vamos introduzir aqui alguns conceitos importantes ao nosso trabalho: *Funções Semânticas*, *Equações Semânticas* e os modelos matemáticos de *Store* (armazenamento) e *Environment* (ambiente).

2.2.1 Funções Semânticas e Equações Semânticas

A semântica de uma linguagem de programação é expressa com base em *Funções Semânticas*. Estas mapeiam a sintaxe (abstrata) de determinado programa a sua *denotação*. As seguintes definições são baseadas em [Wat91]:

- *Funções Semânticas*: Para cada elemento sintático P (frase de um programa) existe uma função f de forma tal que $f: P \rightarrow D$, onde D é o domínio das *denotações* de P ;
- *Equações Semânticas*: Definem efetivamente o significado das partes e subpartes. Dada uma função semântica f , existe uma igualdade $f \llbracket \dots Q \dots R \rrbracket = \dots f'Q \dots f''R \dots$, de maneira que f' e f'' sejam as equações semânticas correspondentes às subfrases Q e R respectivamente.

Os exemplos (2.2.1) e (2.2.2) descrevem estes conceitos em um caso particular.

Exemplo 2.2.1 (Sintaxe abstrata de uma linguagem de expressões)

$$e ::= e + e \mid n$$

A regra anterior determina a estrutura de uma linguagem de números e soma de expressões. A partir deste ponto, será usada uma nova notação (similar a BNF) para a descrição da sintaxe. Esta notação corresponde à utilizada em Semântica de Ações [Mos92]. Por exemplo, a linguagem de expressões acima pode ser descrita por:

$$\text{Expression} ::= \text{Expression} + \text{Expression} \mid \text{Number} \quad (2.5)$$

As variáveis E e N serão usadas para simbolizar elementos de *Expression* e *Number*, respectivamente.

Em Semântica Denotacional expressamos o significado destas regras da seguinte maneira:

Exemplo 2.2.2 (Semântica denotacional da linguagem de expressões)

- $\text{evaluate} : \text{Expression} \rightarrow \text{Natural}$.
 - $\text{sum} : \text{Natural} \times \text{Natural} \rightarrow \text{Natural}$.
- (1) $\text{evaluate} \llbracket E_1:\text{Expression} \text{ "+" } E_2:\text{Expression} \rrbracket = \text{sum}(\text{evaluate } E_1, \text{evaluate } E_2)$
- (2) $\text{evaluate} \llbracket N:\text{Number} \rrbracket = \text{valuation } N$

A função semântica *evaluate* é definida como um mapeamento da entidade sintática *Expression* para sua denotação, que é um número natural representado pelo *sort* (conjunto) *Natural*. A função auxiliar *sum* produz a soma de dois números naturais.

A equação semântica em (1) especifica a soma de expressões. Em (2) definimos a semântica do numeral N como sendo o número *Natural* correspondente. A função auxiliar *valuation* é utilizada para expressar este mapeamento, como em [Wat91].

Linguagens mais complexas, como linguagens imperativas por exemplo, executam operações como declaração de variáveis e atualização de valores em memória. Logo, a definição de um modelo matemático que permita expressar estes comportamentos se faz necessária. Os fundamentos para este fim estão descritos a seguir.

2.2.2 O Modelo de Armazenamento - *Storage*

Definimos uma estrutura de memória, denominada *store*, como um conjunto de *células* ou localizações. Estas células podem estar nos estados [Wat91]:

- *unused*: indica que a célula não está utilizada - está disponível para ser alocada;
- *undefined*: indica que a célula está alocada mas ainda sem valor determinado;
- contendo um valor.

A Figura 2.1 mostra um exemplo de *store*. a primeira célula contém o valor 5. A célula indicada por "?" está no estado *undefined*. As células hachuradas estão no estado *unused*.

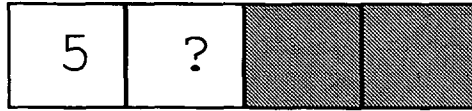


Figura 2.1: Modelo de armazenamento.

Os valores que podem ser armazenados nas células são ditos *storables*. Funções auxiliares são aplicadas ao *store* visando representar operações básicas como alocação de células de memória, procura e atualização de valores em células, etc.

Vejamos a seguir a definição de um comando de atribuição:

Exemplo 2.2.3 (Semântica Denotacional de um comando de atribuição)

- $\text{Storable} : \text{Natural}$
 - $\text{execute} : \text{Command} \rightarrow \text{Store} \rightarrow \text{Store}$.
 - $\text{evaluate} : \text{Expression} \rightarrow \text{Store} \rightarrow \text{Natural}$.
- (1) $\text{execute} \llbracket I:\text{identifier} \text{ ":=" } E:\text{Expression} \rrbracket \text{sto} =$
 $\text{let } val = \text{evaluate } E \text{ sto in update}(\text{sto}, \text{location } I, val)$

No exemplo 2.2.3 definimos um comando de atribuição que atualiza uma célula de memória com o valor resultante da avaliação da expressão E . Os valores que armazenamos nas células (*storables*) são números naturais. A função semântica *execute* define a denotação de comandos, onde um comando atualiza a memória atual produzindo um novo armazenamento. Uma expressão, definida pela função semântica *evaluate*, produz um número natural, lendo (se necessário) as informações da memória corrente.

A equação semântica em (1) *denota* a atribuição. O store *sto* será modificado com o valor *val*, resultante da avaliação da expressão E , na localização referenciada pelo identificador I . Utilizamos aqui a função *location* que indica a célula de memória correspondente ao identificador em questão. A função auxiliar *update* é empregada possibilitando a atualização da memória:

- $\text{location} : \text{Identifier} \rightarrow \text{Location}$.
- $\text{update} : \text{Store} \times \text{Location} \times \text{Storable} \rightarrow \text{Store}$.

A função *update* produz um novo Store, dados o Store atual, uma localização (*Location*) em memória e um valor (*Storable*). Seu objetivo é atualizar a memória com um valor informado.

2.2.3 O Modelo de Ambientes - *Environment*

Muitas linguagens de programação trabalham com o conceito de escopo, que determina a abrangência das variáveis dos seus programas. Desta forma, deve haver uma organização para os nomes de identificadores; considerando não apenas as variáveis, mas também as constantes, os nomes de funções e procedimentos, etc.

Cada declaração em uma linguagem de programação estabelece uma associação de identificadores a entidades da linguagem. Esta ligação denomina-se *binding*. O modelo de ambientes define a existência de um conjunto destes mapeamentos, possibilitando a associação dos identificadores em determinado momento da execução de um programa. Os tipos de valores que podem ser ligados aos identificadores são ditos *bindables*.

Exemplo 2.2.4 (Um ambiente com variáveis e constantes)

$$E = \{ x \mapsto \text{cell1}, y \mapsto \text{true}, z \mapsto 10 \}$$

A notação acima descreve o ambiente E , o qual possui três elementos: uma variável x , representada pelo mapeamento do identificador em questão para uma célula em memória, e as constantes y e z mapeadas para os valores true e 10, respectivamente. A interação com o ambiente é exemplificada a seguir.

De acordo com a mesma idéia aplicada aos *stores*, são disponibilizadas funções auxiliares que permitem criar ambientes, localizar identificadores no ambientes, etc.

Exemplo 2.2.5 (Semântica Denotacional de uma declaração de variável)

- Bindable : Location
 - Storable : Natural
 - 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')

Definimos inicialmente que Bindable são localizações em memória (células), sendo esta definição necessária para efetuar os *bindings* dos identificadores. Os valores a serem armazenados nas células Storable, por sua vez, correspondem a números naturais.

A função semântica elaborate é empregada para especificar a semântica de declarações. Sua definição indica que dada uma declaração (Declaration), o Environ e Store atuais, o resultado é formado por um par contendo um novo Environ - incluindo o mapeamento do identificador para uma célula em memória - e um novo Store - com a célula reservada para a variável. A equação em (1) denota este comportamento.

Uma função auxiliar *bind* é utilizada e tem a seguinte forma:

- $\text{bind} : \text{Identifier} \times \text{Bindable} \rightarrow \text{Environ}$.

Esta, cria um novo ambiente que tem a ligação entre o identificador e o *bindable* correspondente.

Podemos indicar alguns pontos positivos e negativos da semântica denotacional como segue:

- (+) A semântica denotacional é uma técnica bastante avançada para a definição do significado de linguagens de programação;
- (+) Fundamentos matemáticos abstratos podem ser empregados na especificação formal;
- (-) As especificações que utilizam este paradigma são no geral complexas;
- (-) A reutilização e modularidade é dificultada nesta abordagem. A modificação de especificações e adequação para novos trabalhos não é uma tarefa trivial.

Uma visão mais aprofundada do tema que apresentamos nesta Seção pode ser encontrada em [Sch86] e para uma introdução [Wat91].

2.3 Semântica Operacional

Semântica Operacional é uma classe de formalismos onde a descrição de um programa indica *como* este programa é executado. Uma característica padrão de todos os formalismos operacionais é a definição indutiva de uma *relação de transição*, onde é definido o significado das partes sintáticas. Definições indutivas [Acz77] e sistemas de transição [Plo81, Ast91] são utilizados. A introdução que apresentamos nesta Seção está baseada em [Mus96].

2.3.1 Interpretadores de definição

As primeiras abordagens em semântica operacional descreviam a execução de um programa utilizando uma *máquina abstrata*. Como exemplos, podemos citar: SMC [Plo81, Mit93], utilizada principalmente para ensino; SECD [Lan64] e máquinas G [Joh87], utilizadas para a especificação e implementação de linguagens funcionais [Wik87, dLM88, PJ87].

Um SMC é definido como uma transição entre triplas $\langle S, M, C \rangle$, na qual:

S representa uma *pilha de valores*, onde código ou valores podem ser empilhados;

M é o *storage* da máquina, ou seja, um mapeamento entre células de memória e valores (*storable*);

C é a *pilha de controle*, onde o código do programa é empilhado para ser executado pela máquina.

Uma configuração inicial típica para um programa P é $\langle \varepsilon, \phi, P \rangle$, contendo a pilha vazia (ε), o *storage* vazio (ϕ) e o programa P .

A configuração final contém tipicamente um valor na pilha de valores (resultado de um programa), a memória não vazia e a pilha de controle vazia, produzindo uma tripla na forma $\langle n, s, \varepsilon \rangle$.

Mostramos a seguir o funcionamento de uma SMC. A regra a seguir apresenta a sintaxe abstrata de uma linguagem simples de números e adição entre expressões. O símbolo terminal n da gramática representa um número.

Exemplo 2.3.1 (Sintaxe abstrata de uma linguagem de expressões)

$$e ::= n \mid e + e$$

A semântica desta linguagem é definida pelo sistema de transição de estados a seguir:

- (1) $\langle S, M, n \cdot C \rangle \rightarrow \langle n \cdot S, M, C \rangle$
- (2) $\langle S, M, e_1 + e_2 \cdot C \rangle \rightarrow \langle S, M, e_1 \cdot e_2 \cdot "+" \cdot C \rangle$
- (3) $\langle n_2 \cdot n_1 \cdot S, M, "+" \cdot C \rangle \rightarrow \langle \underline{n_1 + n_2} \cdot S, M, C \rangle$

A linha (1) descreve a avaliação de um número. Este é simplesmente transferido da pilha de controle para a pilha de valores.

A linha (2) inicia a avaliação da adição de expressões, dividindo-a em seus componentes. A expressão é avaliada em modo pós-fixado, onde resultados intermediários são armazenados na pilha de valores. Conforme (3), se o símbolo "+" estiver no topo da pilha de controle, os dois números do topo da pilha de valores são somados. O resultado é empilhado na pilha de valores (a notação $\underline{n_1 + n_2}$ representa o *resultado* da adição de dois números n_1 e n_2).

Para uma visão mais aprofundada de SMC consultar [Plo81, Mit93].

2.3.2 Semântica de Transição

Plotkin, em [Plo81], introduziu inicialmente a Semântica Operacional Estrutural (*Structural Operational Semantics*). Este formalismo permite a definição da relação de um *passo* (*step*) entre partes da linguagem especificada. A semântica completa do programa é definida por uma relação de *passo* e seu fechamento transitivo e reflexivo. Em um trabalho complementar, Astesiano [Ast91] generalizou o formalismo da Semântica Operacional Estrutural, criando a Semântica Estrutural Indutiva (*Structural Inductive Semantics*).

A semântica definida em [Plo81] corresponde a Semântica Estrutural Indutiva de "passo-curto" (*small-step*), sendo referenciada como Semântica de Transição (*Transition Semantics*) em [Gun91]. Nesta Seção vamos apresentar brevemente esta abordagem com base em exemplos.

A seguinte regra apresenta a sintaxe abstrata do exemplo 2.3.1, descrita na Seção anterior:

$$e ::= n \mid e + e \quad (2.6)$$

O terminal n , representa um número. Para definir a semântica operacional de uma frase em semântica de transição utilizamos *regras*. Uma regra é definida para cada passo possível de um programa dado e a definição destes passos está baseada na estrutura sintática da linguagem. A semântica de transição da pequena linguagem de expressões (2.6) pode ser dada pelas seguintes regras:

$$\frac{e_1 \rightarrow e'_1}{e_1 + e_2 \rightarrow e'_1 + e_2} \quad \frac{e_2 \rightarrow e'_2}{e_1 + e_2 \rightarrow e_1 + e'_2} \quad \frac{}{n_1 + n_2 \rightarrow \underline{n_1 + n_2}}$$

Estas regras indicam que os lados de uma adição de expressões serão avaliados até que sejam reduzidos a números. O resultado da expressão é obtido pela adição destes números. Devemos ressaltar alguns pontos nesta especificação simples:

- Não há regra semântica para os números, uma vez que estes estão no estado final (do ponto de vista da relação de passo);
- Não existe uma ordem fixada para aplicar as regras sobre expressões. Qualquer ordem de avaliação pode ser definida aqui, apenas modificando as regras para o caminho desejado (para o caso de impor avaliação da esquerda para a direita, por exemplo);
- Alguns itens sintáticos são tratados como valores semânticos. A expressão $\underline{n_1 + n_2}$ deve ser vista como um número representando a soma de n_1 e n_2 .

2.3.2.1 Uma linguagem imperativa simples

A gramática a seguir mostra uma linguagem com expressões e comandos. As expressões são similares às da linguagem anterior, porém variáveis “ x ” estão adicionadas. Os comandos da linguagem incluem: declaração de variáveis, atribuições, comando nulo, seleção simples, repetição e seqüenciação.

Exemplo 2.3.2 (Sintaxe abstrata de uma linguagem imperativa simples)

$$e ::= n \mid x \mid e + e$$

$$c ::= \text{var } x \mid x := e \mid \text{skip} \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c \mid c; c$$

O comando **skip** representa a operação nula. A introdução deste comando simplifica a especificação dos comandos, definindo a noção de *terminação* da computação de uma frase. Este ponto será melhor esclarecido ao final desta Seção.

Incluimos aqui as noções de *storage* e *bindings* na semântica dinâmica¹. Estes elementos, em conjunção com a reescrita da frase sintática, vão representar o *estado* do programa.

O estado *final* é o estado onde regras não são mais aplicadas. Um *store* será definido como um mapeamento de células de memória para números. *Bindings*, no programa, serão representados por mapeamentos de variáveis para células, como segue:

$$s : S = [k \mapsto n]$$

$$b : B = [x \mapsto k]$$

$$k : K = \square \text{ (Células de Memória)}$$

A notação “ $\square$ ” [Mos92] indica que o conjunto de células de memória K está indefinido. As relações de transição são definidas sobre triplas $\langle s, b, e \rangle$ no caso de expressões, e triplas $\langle s, b, c \rangle$ no caso de comandos. A relação de um passo para expressões é dada pelas regras a seguir.

¹*Semântica Dinâmica* é aquela que descreve o comportamento dos programas na sua execução. Este termo é utilizado em contraste com a definição da *Semântica Estática*, que descreve as restrições de contexto das partes de um programa, como por exemplo, a semântica dos tipos da linguagem.

$$\boxed{_ \rightarrow _ :: \langle S \times B \times e \rangle \times \langle S \times B \times e \rangle}$$

$$\overline{\langle s, b, x \rangle \rightarrow \langle s, b, s(b(x)) \rangle} \quad (2.7)$$

$$\frac{\langle s, b, e_1 \rangle \rightarrow \langle s', b', e'_1 \rangle}{\langle s, b, e_1 + e_2 \rangle \rightarrow \langle s', b', e'_1 + e_2 \rangle} \quad (2.8)$$

$$\frac{\langle s, b, e_2 \rangle \rightarrow \langle s', b', e'_2 \rangle}{\langle s, b, e_1 + e_2 \rangle \rightarrow \langle s', b', e'_1 + e'_2 \rangle} \quad (2.9)$$

$$\overline{\langle s, b, n_1 + n_2 \rangle \rightarrow \langle s, b, \underline{n_1 + n_2} \rangle} \quad (2.10)$$

É importante ressaltarmos que, para estas regras, ambos os lados da relação de transição estão construídos a partir do mesmo conjunto de tuplas. Este comportamento é definido pelo quadro anterior à regra (2.7), que estabelece a aridade da relação.

A regra (2.7) define a semântica de uma variável na expressão. A variável é substituída pelo valor correspondente, a partir da memória e ambiente atuais.

As regras (2.8), (2.9) e (2.10) são similares àquelas apresentadas na linguagem de expressões: cada lado de uma soma é avaliado para retornar dois números, que são adicionados, fornecendo o resultado final.

A relação de transição para comandos é definida pelas regras (2.11) a (2.19). A regra (2.11) especifica a declaração de variáveis. Uma nova célula é alocada - não presente no domínio ($\mathbf{Dom}(s)$) de s - e os mapeamentos de *store* e *bindings* são estendidos para conter a nova variável. O valor 0 é armazenado na nova célula. A notação $b[x/k]$ denota o mapeamento:

$$b[x/k](u) = \begin{cases} b(u) & \text{if } u \neq x \\ k & \text{if } u = x \end{cases}$$

E uma notação similar é utilizada para stores.

$$\boxed{_ \rightarrow _ :: \langle S \times B \times c \rangle \times \langle S \times B \times c \rangle}$$

$$\overline{\langle s, b, \mathbf{var } x \rangle \rightarrow \langle s[k/0], b[x/k], \mathbf{skip} \rangle} \text{ (onde } k \notin \mathbf{Dom}(s) \text{)} \quad (2.11)$$

$$\frac{\langle s, b, e \rangle \rightarrow \langle s', b', e' \rangle}{\langle s, b, x := e \rangle \rightarrow \langle s', b', x := e' \rangle} \quad (2.12)$$

$$\overline{\langle s, b, x := n \rangle \rightarrow \langle s[b(x)/n], b, \mathbf{skip} \rangle} \quad (2.13)$$

$$\frac{\langle s, b, e \rangle \rightarrow \langle s, b, e' \rangle}{\langle s, b, \mathbf{if } e \mathbf{ then } c_1 \mathbf{ else } c_2 \rangle \rightarrow \langle s, b, \mathbf{if } e' \mathbf{ then } c_1 \mathbf{ else } c_2 \rangle} \quad (2.14)$$

$$\frac{\langle s, b, e \rangle \rightarrow \langle s, b, 1 \rangle}{\langle s, b, \mathbf{if } e \mathbf{ then } c_1 \mathbf{ else } c_2 \rangle \rightarrow \langle s, b, c_1 \rangle} \quad (2.15)$$

$$\frac{\langle s, b, e \rangle \rightarrow \langle s, b, 0 \rangle}{\langle s, b, \text{if } e \text{ then } c_1 \text{ else } c_2 \rangle \rightarrow \langle s, b, c_2 \rangle} \quad (2.16)$$

$$\overline{\langle s, b, \text{while } e \text{ do } c \rangle \rightarrow \langle s, b, \text{if } e \text{ then } (c ; \text{while } e \text{ do } c) \text{ else skip} \rangle} \quad (2.17)$$

$$\frac{\langle s, b, c_1 \rangle \rightarrow \langle s', b', c'_1 \rangle}{\langle s, b, c_1; c_2 \rangle \rightarrow \langle s', b', c'_1; c_2 \rangle} \quad (2.18)$$

$$\overline{\langle s, b, \text{skip}; c_2 \rangle \rightarrow \langle s, b, c_2 \rangle} \quad (2.19)$$

As regras (2.12) e (2.13) descrevem o significado de atribuições. A expressão do lado direito do comando é avaliada até que um número seja obtido, sendo este número armazenado na célula correspondente ao seu identificador, utilizando o *store* e *environment* atuais.

Definimos de (2.14) a (2.16) a semântica da estrutura de seleção. A avaliação da expressão é definida em (2.14). Caso a avaliação resulte no valor 1, o primeiro comando é executado, conforme (2.15). Caso a avaliação da expressão resulte em 0, o segundo comando é executado (2.16).

A regra (2.17) define o significado da estrutura de repetição. A semântica desta estrutura é expressa com base na estrutura de seleção previamente descrita. Caso a expressão avaliada resulte em 1, o comando é executado e o teste se repete até que resulte em 0, e (**skip**) seja obtido.

Em (2.18) e (2.19) definimos o significado das seqüências de comandos. O primeiro comando é executado até que o comando nulo (**skip**) é obtido. Em seguida os demais são executados.

Na descrição acima, não há passos possíveis começando por configurações na forma $\langle s, b, \text{skip} \rangle$, caracterizando as configurações finais. A noção de *resultado* pode ser definida em termos destas configurações finais:

O conjunto de mapeamentos

$$c \downarrow^{s,b} = \{s' \cdot b' \mid \langle s, b, c \rangle \rightarrow^* \langle s', b', \text{skip} \rangle\}$$

onde “ $_ \cdot _$ ” é a composição de funções, caracteriza todos os possíveis resultados de um programa c , dados os *bindings* iniciais b e o *store* s . O conjunto $c \downarrow^{s,b}$ é constituído por todos os mapeamentos relacionando identificadores aos seus valores após a execução de c . Da mesma forma que **skip** é utilizado para indicar o término normal da execução do programa, outros tipos de finalizadores podem ser definidos para condições especiais.

Por exemplo, na semântica de transição da notação de ações (*Action Notation*) de [Mos92], três diferentes alternativas de término são definidas, caracterizando término normal, falha e

término excepcional. Isto é dado pela adição de uma nova sintaxe para cada uma destas possíveis condições de terminação. É possível, todavia, evitar o uso do **skip** para indicar o término. Assim, uma definição alternativa da relação de transição pode ser utilizada [Mus96].

Algumas conclusões positivas (+) e negativas (−) podem ser obtidas das regras anteriores:

- (+) As regras semânticas são, no geral, simples e fáceis de entender;
- (+) A simplicidade da Semântica de Transição fornece uma grande variedade de estilos de especificação. Existe um número de caminhos possíveis para se especificar o mesmo construtor. Entidades sintáticas e semânticas podem ser facilmente combinadas;
- (−) O número de regras necessárias para definir uma linguagem é geralmente grande, quando comparado com o número de regras sintáticas que definem a linguagem. No caso de nossa linguagem simples, treze regras semânticas são necessárias para definir a semântica de nove regras de produção. Para linguagens reais, o tamanho da especificação semântica é muito maior que a descrição sintática;
- (−) As descrições não são composicionais, em geral. A semântica de uma frase não é dada apenas pela composição da semântica de suas sub-frases, como acontece na regra (2.17).

2.3.2.2 Semântica de passo misto

Em Semântica de Transição o fechamento transitivo e reflexivo da relação de um passo pode ser utilizada para definir a semântica de construtores sintáticos. Por exemplo, no caso da linguagem anterior, uma semântica alternativa para comandos pode ser definida:

$$\boxed{_ \rightarrow _ :: \langle s \times b \times c \rangle \times \langle s \times b \times c \rangle}$$

$$\overline{\langle s, b, \text{var } x \rangle \rightarrow \langle s[k/0], b[x/k], \text{skip} \rangle} \quad (\text{where } k \notin \text{Dom}(s))$$

$$\frac{\langle s, b, e \rangle \rightarrow^* \langle s', b', n \rangle}{\langle s, b, x := e \rangle \rightarrow \langle s[b(x)/n], b, \text{skip} \rangle} \quad (2.20)$$

$$\frac{\langle s, b, c_1 \rangle \rightarrow^* \langle s', b', \text{skip} \rangle \quad \langle s', b', c_2 \rangle \rightarrow^* \langle s'', b'', \text{skip} \rangle}{\langle s, b, c_1; c_2 \rangle \rightarrow \langle s'', b'', \text{skip} \rangle} \quad (2.21)$$

$$\frac{\langle s, b, e \rangle \rightarrow^* \langle s, b, 1 \rangle}{\langle s, b, \text{if } e \text{ then } c_1 \text{ else } c_2 \rangle \rightarrow \langle s, b, c_1 \rangle} \quad (2.22)$$

$$\frac{\langle s, b, e \rangle \rightarrow^* \langle s, b, 0 \rangle}{\langle s, b, \text{if } e \text{ then } c_1 \text{ else } c_2 \rangle \rightarrow \langle s, b, c_2 \rangle} \quad (2.23)$$

$$\frac{\langle s, b, e \rangle \rightarrow^* \langle s, b, 0 \rangle}{\langle s, b, \text{while } e \text{ do } c \rangle \rightarrow \langle s, b, \text{skip} \rangle} \quad (2.24)$$

$$\frac{\langle s, b, e \rangle \rightarrow^* \langle s, b, 1 \rangle \quad \langle s, b, c \rangle \rightarrow^* \langle s', b', \text{skip} \rangle}{\langle s, b, \text{while } e \text{ do } c \rangle \rightarrow \langle s', b', \text{while } e \text{ do } c \rangle} \quad (2.25)$$

Na regra (2.20), o fechamento transitivo da relação de passo para expressões é utilizado para definir o significado das atribuições, enquanto que na regra (2.21), estados finais são vistos nas premissas, representando o construtor de seqüenciação.

Em ambas, as premissas indicam que a avaliação delas é atômica. Esta é uma restrição à semântica prévia sobre comandos. Este estilo de especificação em Semântica de Transição é denominado *passo-misto* ou “mixed-step” *Structural Inductive Semantics* segundo [Ast91].

As regras (2.22) e (2.23) especificam o comando de seleção. O fechamento transitivo do passo da relação é utilizado nas premissas de ambos os casos, determinando quais dos comandos devem ser executados: c_1 , no caso da expressão resultar em 0; e c_2 , se a expressão resultar em 1.

O comando de repetição é determinado pelas regras (2.24) e (2.25). A primeira contempla o caso do término da iteração, quando a expressão indicada na premissa resulta no valor 0. A segunda estabelece a re-execução, quando o resultado da expressão corresponde a 1 e o comando a ser executado por um ou mais passos produz **skip**.

As especificações neste estilo têm tamanho reduzido e um novo conceito de notação é definido ($\rightarrow^*$), sendo levado em consideração na descrição das árvores de prova e nas provas de propriedades da especificação [Mus96].

2.3.3 Semântica Relacional

A idéia central em Semântica Relacional (*Relational Semantics*) [dS92][Win93] é definir uma relação direta entre as frases sintáticas e seus significados. Neste caso, a definição passo a passo da semântica de transição é substituída por um passo simples contendo frases sintáticas, dados e resultado. Vamos exemplificar esta abordagem, descrevendo a linguagem simples de expressões da Seção anterior:

$$e ::= n \mid e + e$$

Para cada regra sintática da gramática, uma regra semântica é definida, relacionando a expressão sintática ao resultado de sua avaliação:

$$\frac{}{n \triangleright n}$$

$$\frac{e_1 \triangleright n_1 \quad e_2 \triangleright n_2}{e_1 + e_2 \triangleright \underline{n_1 + n_2}}$$

A primeira regra indica que a avaliação de um número é ele mesmo, enquanto que a segunda indica que a avaliação de uma adição entre expressões é a soma dos resultados da avaliação de cada um dos termos. Entidades semânticas podem ser adicionadas à especificação e refletir, por exemplo, no ambiente em que cada expressão deve ser avaliada.

Introduzindo variáveis às expressões da linguagem anterior e adicionando as mesmas notações de *store* e *bindings* da Seção 2.3.2.1, podemos definir as regras para a semântica relacional das expressões como segue:

$$e ::= n \mid x \mid e + e \tag{2.26}$$

$$\boxed{_, _ \vdash _ \triangleright _ :: S \times B \times e \times n}$$

$$\frac{}{s, b \vdash n \triangleright n} \tag{2.27}$$

$$\frac{}{s, b \vdash x \triangleright s(b(x))} \tag{2.28}$$

$$\frac{s, b \vdash e_1 \triangleright n_1 \quad s, b \vdash e_2 \triangleright n_2}{s, b \vdash e_1 + e_2 \triangleright \underline{n_1 + n_2}} \tag{2.29}$$

O quadro inicial indica a aridade da relação que está sendo definida. A regra (2.27) indica que um número resulta nele mesmo em qualquer *storage* e *environment*. A avaliação de identificadores é definida pela regra (2.28). O resultado da expressão é o valor armazenado na célula indicada pelo ambiente *b*. O significado da regra (2.29) é similar ao do exemplo anterior.

2.3.3.1 Uma linguagem imperativa simples

Vamos descrever a seguir a semântica relacional da linguagem imperativa introduzida pelo exemplo 2.3.3, contendo declaração de variáveis, atribuições, estruturas de seleção e repetição, além de seqüências de comandos. A linguagem utiliza o não-terminal *e* de acordo com as expressões definidas pela regra (2.26) anterior.

Exemplo 2.3.3 (Sintaxe abstrata de uma linguagem imperativa simples)

$$e ::= n \mid x \mid e + e$$

$$c ::= \text{var } x \mid x := e \mid \text{skip} \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c \mid c; c$$

As equações semânticas para esta linguagem são como segue:

$$\boxed{_,_ \vdash _ \triangleright _,_ :: S \times B \times c \times S \times B}$$

$$\frac{}{s, b \vdash \text{var } x \triangleright s[k/0], b[x/k]} \quad (\text{where } k \notin \text{Dom}(s)) \quad (2.30)$$

$$\frac{}{s, b \vdash \text{skip} \triangleright s, b} \quad (2.31)$$

$$\frac{s, b \vdash e \triangleright n \quad b(x) = k}{s, b \vdash x := e \triangleright s[k/n], b} \quad (2.32)$$

$$\frac{s, b \vdash e \triangleright 1 \quad s, b \vdash c_1 \triangleright s', b'}{s, b \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 \triangleright s', b'} \quad (2.33)$$

$$\frac{s, b \vdash e \triangleright 0 \quad s, b \vdash c_2 \triangleright s', b'}{s, b \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 \triangleright s', b'} \quad (2.34)$$

$$\frac{s, b \vdash e \triangleright 0}{s, b \vdash \text{while } e \text{ do } c \triangleright s, b} \quad (2.35)$$

$$\frac{s, b \vdash e \triangleright 1 \quad s, b \vdash c \triangleright s', b' \quad s', b' \vdash \text{while } e \text{ do } c \triangleright s'', b''}{s, b \vdash \text{while } e \text{ do } c \triangleright s'', b''} \quad (2.36)$$

$$\frac{s, b \vdash c_1 \triangleright s', b' \quad s', b' \vdash c_2 \triangleright s'', b''}{s, b \vdash c_1; c_2 \triangleright s'', b''} \quad (2.37)$$

O significado de uma declaração de variáveis é dado pela regra (2.30). Da mesma forma apresentada na Semântica de Transição, uma nova célula de memória é reservada e inicializada com zero. O mapeamento de *bindings* é modificado, relacionando a variável x à nova célula. O significado do comando **skip** é dado pela regra (2.31), não havendo qualquer modificação de *store* ou *bindings* neste caso.

O significado da atribuição é dado pela regra (2.32), onde o valor obtido da avaliação da expressão é armazenado na célula k , referenciada pelo identificador x .

As regras (2.33) e (2.34) definem a estrutura de seleção. Caso o valor resultante da avaliação da expressão seja 1, o comando c_1 será executado e a memória e ambiente finais serão determinados por este comando. Da mesma forma, o resultado final de ambiente e memória será determinado pela execução de c_2 , caso o resultado seja 0.

A estrutura de repetição é determinada por (2.35) e (2.36). A regra (2.35) determina que a memória e ambientes não serão modificados no caso em que a avaliação da expressão resultar em 0. Caso o resultado seja 1, o comando é executado produzindo nova memória e ambiente. Estes serão repassados recursivamente à estrutura de repetição que produzirá o resultado final, conforme (2.36).

A regra (2.37) representa a composição de comandos: o primeiro comando (c_1) é executado resultando em um novo par de *store* e *bindings*. Estes por sua vez são repassados para o segundo comando (c_2), para proceder sua execução.

Vamos resumir a seguir as principais características da semântica relacional, ressaltando os pontos positivos e negativos:

- (+) A definição de uma relação contendo o valor inicial e final da computação contribui para a facilidade e uso intuitivo deste formalismo;
- (+) As regras semânticas são no geral simples e fáceis de entender;
- (+) O número de regras semânticas para um linguagem dada é, em geral, similar ao número de regras sintáticas definindo a mesma linguagem;
- (-) As descrições resultantes não são composicionais. A semântica de uma frase dada não é composta apenas pela semântica de suas sub-frases.

CAPÍTULO 3

SEMÂNTICA DE AÇÕES

Em Semântica Denotacional [Wat91] [Sch86], a denotação de qualquer frase é obtida por um mapeamento. Na prática, a definição destes mapeamentos é por vezes complexa, resultando em uma especificação de difícil entendimento, extensão e modificação. Esta realidade serviu de motivação para o aperfeiçoamento e a criação técnicas que melhorem a legibilidade de especificações. Um esforço neste sentido tem a denominação Semântica de Ações.

3.1 Introdução

Semântica de Ações foi desenvolvida por Peter Mosses [Mos92] com a colaboração de David Watt [Wat91]. As especificações nesta abordagem utilizam a mesma estrutura da Semântica Denotacional [Wat91] [Sch86], onde o significado das partes de uma linguagem é apresentado com base em *funções semânticas* e *equações semânticas*. Uma notação especial denominada Notação de Ações (*Action Notation*) foi desenvolvida, em substituição à notação funcional utilizada na semântica denotacional, sendo base para as especificações neste paradigma.

3.2 Notação de Dados

Em Semântica de Ações, números naturais, números inteiros, caracteres, listas, strings, tuplas, mapeamentos, árvores, etc., são classificados como *sorts*. Podemos defini-los como conjuntos de elementos com *operações* relacionadas, sendo que a coleção destes conjuntos forma a Notação de Dados (*Data Notation*) em Semântica de Ações.

Exemplo 3.2.1 (Notação padrão de dados)

1, true: correspondem a valores simples;

truth-value, integer, cell[integer]: são exemplos de *sorts* criados para representar respectivamente: valores-verdade, valores inteiros e células de memória que armazenam valores inteiros;

`sum(1,5)`, `less(3,5)`: representam operações. A operação `sum` resulta no inteiro (6) correspondente a soma dos valores indicados. A operação `less` resulta em um valor-verdade (`true`) determinando que o primeiro parâmetro é menor do que o segundo.

Dados podem ser produzidos durante a execução de uma ação por uma entidade especial denominada *yielder*. Baseado na entrada informada, esta entidade é avaliada produzindo uma nova informação. Detalharemos melhor este conceito nas Seções seguintes.

A descrição completa da notação padrão de dados pode ser encontrada no apêndice B de [Mos92].

3.3 Notação de Ações

Em Semântica de Ações a *denotação* de determinada frase é uma *ação*. Ações são entidades que recebem e propagam diversos tipos de dados e que podem ser executadas. Ações, quando executadas, podem produzir os seguintes resultados:

complete: indicando término normal - a execução procedeu normalmente;

escape: indica término anormal - resultado inesperado;

fail: a execução resultou em falha;

diverge: não finalizou;

As ações podem ser classificadas como *primitivas* e *compostas*. As ações primitivas são executadas *atomicamente* produzindo os resultados previamente descritos. As seguintes ações são primitivas:

complete: completa imediatamente;

fail: falha imediatamente;

diverge: não finaliza;

Ações compostas são criadas com o uso de combinadores de ações (*action combinators*). Estas podem ser: *seqüenciadas* ou seja, uma sub-ação é executada antes de outra; *intercaladas*, as suas sub-ações atômicas são executadas em ordem aleatória; *escolhidas exclusivamente*, onde apenas uma das sub-ações é escolhida para execução [Mou93].

As ações podem manipular (receber e produzir) diversos tipos de dados estabelecendo um fluxo. Os dados transitórios (*transient information*) são passados imediatamente entre as ações.

Os *bindings* (*scoped information*) disponíveis a uma ação podem ser repassados às sub-ações e as informações disponíveis em células (*storable information*) podem ser lidas e modificadas.

Exemplo 3.3.1 (Fluxo de dados em Semântica de Ações)

$\{0 \mapsto 31, 3 \mapsto \text{false}, 5 \mapsto \text{cell1}\}$ constituem dados transitórios. São representados por mapeamentos de números (*labels*) para dados;

$\{“x” \mapsto \text{true}, “y” \mapsto \text{cell2}, “z” \mapsto 10\}$ são os *bindings*. É um conjunto de mapeamentos de identificadores para dados, ditos *bindables*;

$\{\text{cell1} \mapsto \text{true}, \text{cell2} \mapsto \text{false}, \text{cell3} \mapsto \text{cell1}\}$ é um mapeamento entre células e dados denominados *storables*.

Baseados então nestes tipos de informação podemos classificar as ações e combinadores de acordo com as seguintes *facetas*:

- *faceta básica*: esta faceta trata do fluxo de controle;
- *faceta funcional*: corresponde às ações que processam dados transitórios (recebem ou fornecem dados);
- *faceta declarativa*: engloba as ações que recebem ou geram *bindings*;
- *faceta imperativa*: ações que persistem ou consultam informações em memória;

A descrição apresentada na Seção seguinte traz uma breve introdução das facetas das ações baseada em [Mos92], [Wat91], [Mou93] e [Bro96]. A linguagem imperativa simples, mostrada no exemplo 2.3.2 da Seção 2.3.2.1, será utilizada para exemplificarmos esta abordagem.

3.3.1 Faceta Básica

Corresponde à representação da ordem execução das sub-ações de uma ação composta. Na faceta básica, estabelecemos, por exemplo, se as ações devem ser seqüenciadas ou intercaladas. Uma ação primitiva *complete* finaliza normalmente, sem produzir *bindings* ou modificar o *storage*. Todavia, ações construídas com o uso dos *action combinators* possuem comportamento diferenciado. Vamos observar as seguintes ações:

- $A_1 \text{ or } A_2$;
- $A_1 \text{ and then } A_2$;
- $A_1 \text{ and } A_2$.

A ação composta A_1 or A_2 estabelece a escolha não-determinística de *uma* das sub-ações para ser executada. Porém se ação escolhida falhar a outra é executada. A ação composta falha, se *ambas* falharem, e completa, se uma delas completa.

A ação composta A_1 and then A_2 estabelece a execução *seqüencial* das ações. Os dados transitórios e *bindings* de entrada são distribuídos às ações e as informações de resultado são *mescladas* gerando um novo conjunto de dados transitórios e *bindings*.

As sub-ações de A_1 and A_2 são intercaladas, ou seja, a escolha da ordem de execução é não-determinística. O fluxo dos dados transitórios e *bindings* de entrada é igual ao da ação anterior, porém o *store* final é resultado da execução da segunda sub-ação.

Exemplo 3.3.2 (Combinadores básicos)

(1) execute $\llbracket C_1:\text{Command} \text{ " ; " } C_2:\text{Command} \rrbracket =$
 | execute C_1
 and then
 | execute C_2

Em (1) exemplificamos a execução seqüencial de C_1 e C_2 pelo uso do combinador and then. Alguns combinadores, todavia, são utilizados especificamente sobre ações de determinadas facetas. Nas Seções seguintes, introduziremos alguns deles e apresentaremos algumas dessas ações.

3.3.2 Faceta Funcional

Esta faceta engloba as ações que trabalham com dados transitórios. Basicamente, estas ações recebem dados de ações e fornecem novos dados para outras. A ação primitiva desta faceta está representada a seguir:

“give Y ”

Esta ação produz um dado simples, resultado da avaliação de Y , identificado pelo rótulo 0.

Exemplo 3.3.3 (Fluxo de dados transitórios)

Com base nos dados transitórios $\{ 1 \mapsto \text{false}, 2 \mapsto 15, 3 \mapsto \text{true} \}$:

“give the truth-value #3” produz o dado transitório $\{ 0 \mapsto \text{true} \}$;

“give the truth-value #1” produz o dado transitório $\{ 0 \mapsto \text{false} \}$;

“give the integer #2” produz o dado transitório $\{ 0 \mapsto 15 \}$.

O combinador funcional “then” estabelece a seqüenciação e transferência de dados transitórios entre as ações. A_1 then A_2 define que A_1 deve ser executada recebendo os dados transitórios de entrada. O resultado produzido por A_1 é então repassado para a execução de A_2 . Vamos exemplificar o uso desta notação em uma soma de expressões.

Exemplo 3.3.4 (Ações e combinadores funcionais)

(1) evaluate $\llbracket E_1:\text{Expression} \text{ “+” } E_2:\text{Expression} \rrbracket =$
 $\begin{array}{|l} \text{evaluate } E_1 \\ \text{and} \\ \text{evaluate } E_2 \\ \text{then} \\ \text{give sum(the given number \#0, the given number \#1)} \end{array}$

As expressões E_1 e E_2 são avaliadas produzindo dados rotulados por 1 e 2 respectivamente. Esta *tupla* de dados é repassada à última sub-ação que produz a soma dos dados.

3.3.3 Faceta Imperativa

As ações da faceta imperativa interagem com a memória (*storage*). Um conjunto de células é disponibilizado, permitindo assim o armazenamento de informações. Ações imperativas podem: ler o *store* atual e efetuar modificações pela alocação e liberação de células e pelo armazenamento de valores nas células alocadas. A ação utilizada para armazenar valores no *store* é:

“store Y_1 in Y_2 ”

O *yielder* Y_1 é avaliado produzindo um dado simples. A célula resultante da avaliação de Y_2 recebe então o dado, finalizando a ação. Caso a célula não esteja alocada, a ação falha.

Exemplo 3.3.5 (Fluxo de informações em memória)

Dado o seguinte *storage* $\{ \text{cell1} \mapsto 10, \text{cell2} \mapsto \text{true}, \text{cell3} \mapsto \text{cell1} \}$:

“store false in cell2” produz um novo *store* representado por $\{ \text{cell1} \mapsto 10, \text{cell2} \mapsto \text{false}, \text{cell3} \mapsto \text{cell1} \}$, que contém o valor (*storable*) false na célula cell2;

“deallocate cell2” desaloca a célula cell2;

“give the integer stored in cell1” retorna o valor 10.

A ação *store* pode ser utilizada na representação de comandos de atribuição, como no exemplo 3.3.6.

Exemplo 3.3.6 (Ações e combinadores imperativos)

```
(1) execute [ I:identifier "!=" E:Expression ] =  
    | evaluate E  
    then  
    | store the given number in the cell bound to I
```

O valor resultado da avaliação de E é repassado a segunda sub-ação, que é responsável por armazená-lo na célula referenciada pelo identificador I .

Uma ação primitiva nesta faceta é “allocate Y ”. Esta aloca arbitrariamente uma célula no *store* corrente com estado *undefined* [Wat91], indicando que a célula está reservada mas ainda sem valor. Esta ação possui comportamento diferenciado pois, além de executar a operação descrita, ela *produz* um dado que é a célula propriamente dita (envolvendo as facetas funcional e imperativa). Este tipo de ação é denominada *híbrida* ou multi-facetada. Ações *híbridas* assim como a faceta *reflexiva* serão brevemente abordadas no Capítulo 5.

3.3.4 Faceta Declarativa

A faceta declarativa engloba as ações que definem *bindings*. Um *binding* é um mapeamento entre identificadores e *bindables* (elementos do *sort* *bindable*). Dada a seguinte ação:

“bind k to Y ”

O *yielder* Y é avaliado produzindo um dado simples. Um *binding* simples então é gerado, ligando o identificador k ao dado resultante.

Exemplo 3.3.7 (Fluxo de *Bindings*)

Dado o conjunto de bindings $\{x \mapsto 10, y \mapsto \text{true}, z \mapsto \text{cell1}\}$:

“bind x to 2” produz o um novo ambiente (*environment*) na forma $\{x \mapsto 2\}$;

“rebind” repassa os bindings atuais resultando em $\{x \mapsto 10, y \mapsto \text{true}, z \mapsto \text{cell1}\}$;

“give the integer bound to “y”” falha, pois o identificador y não corresponde a um valor inteiro.

A ação composta A_1 hence A_2 estabelece que os *bindings* definidos pela sub-ação A_1 serão repassados à ação A_2 .

Os *bindings* passados para a ação composta A_1 moreover A_2 são distribuídos entre as subações. Os *bindings* resultantes, todavia, são uma composição dos resultados de A_1 e A_2 , sendo que os de A_2 têm prioridade em caso de coincidência.

A ação furthermore A produz os bindings recebidos, atualizados com os bindings da ação A .

O seguinte comando cria *bindings*:

Exemplo 3.3.8 (Ações e combinadores declarativos)

```
(1) execute [ [ "var" I:Identifier ] =
      | allocate a cell
      | then
      | bind I to the cell #0
      | hence
      | store 0 in the cell bound to I
```

Uma célula de memória é alocada e repassada com o rótulo 1. Um novo ambiente é criado contendo o *binding* do identificador à célula. Finalmente o valor 0 é armazenado na célula referenciada pelo identificador.

3.3.5 Exemplo da linguagem imperativa simples

Vamos definir a semântica da linguagem imperativa simples introduzida no exemplo 2.3.2 da Seção 2.3.2.1. A primeira parte desta descrição contém a sintaxe abstrata da linguagem e a na sequência, a semântica dinâmica da mesma, composta pelas entidades semânticas e funções semânticas.

3.3.5.1 Sintaxe abstrata

Esta descrição utiliza a notação padrão de dados definida por [Mos92]. Símbolos não-terminais (lado esquerdo das definições) são utilizados como termos de cada equação. Os colchetes duplos ($[\dots]$) definem as árvores sintáticas e os símbolos terminais são identificados pelas aspas duplas.

grammar:

- Expression = Number | Identifier | [Expression "+" Expression] .
- Command = ["var" Identifier] | ["skip"] | [Identifier ":" "=" Expression] | ["if" Expression "then" Command "else" Command] | ["while" Expression "do" Command] | [Command ";" Command].

3.3.5.2 Entidades semânticas

A definição para esta linguagem é simples uma vez que não utilizamos *sorts* de dados complexos. Definimos a seguir *number* correspondendo aos números naturais, sendo que estes são os valores que podem ser armazenados em células (*storable*). As células, por sua vez, são os dados que podem ser ligados (*bindable*) aos identificadores produzindo os *bindings*.

- *number* = *natural* .
- *storable* = *number* .
- *bindable* = *cell* .

3.3.5.3 Funções semânticas

As funções semânticas utilizadas são unicamente *evaluate* e *execute*. Estas mapeiam, respectivamente, expressões e comandos aos seus significados.

introduces: *evaluate* __ , *execute* __ .

- *evaluate* __ :: *Expression* → *action* .
- (1) *evaluate* [*N*:*Number*] =
 | give valuation of *N*
 - (2) *evaluate* [*I*:*identifier*] =
 | give the number bound to *I*
 - (3) *evaluate* [*E*₁:*Expression* "+" *E*₂:*Expression*] =
 | evaluate *E*₁
 | and
 | evaluate *E*₂
 | then
 | give sum(the given number #0, the given number #1)
- *execute* __ :: *Command* → *action* .
- (4) *execute* ["var" *I*:*Identifier*] =
 | allocate a cell
 | then
 | bind *I* to the cell #0
 | hence
 | store 0 in the cell bound to *I*
 - (5) *execute* [*I*:*identifier* "!=" *E*:*Expression*] =
 | evaluate *E*
 | then
 | store the given number in the cell bound to *I*
 - (6) *execute* ["skip"] =
 | complete

- (7) $\text{execute } \llbracket \text{"if" } E:\text{Expression "then" } C_1:\text{Command "else" } C_2:\text{Command} \rrbracket =$
 $\quad \left| \begin{array}{l} \text{evaluate } E \\ \text{then} \\ \left| \begin{array}{l} \text{check (the given number is 1) and then execute } C_1 \\ \text{or} \\ \text{check (the given number is 0) and then execute } C_2 \end{array} \right. \end{array} \right.$
- (8) $\text{execute } \llbracket \text{"while" } E:\text{Expression "do" } C:\text{Command} \rrbracket =$
 $\quad \text{unfolding}$
 $\quad \left| \begin{array}{l} \text{evaluate } E \\ \text{then} \\ \left| \begin{array}{l} \text{check (the given number is 1) and then execute } C \text{ and the unfold} \\ \text{or} \\ \text{check (the given number is 0) and then complete} \end{array} \right. \end{array} \right.$
- (9) $\text{execute } \llbracket C_1:\text{Command ";" } C_2:\text{Command} \rrbracket =$
 $\quad \left| \begin{array}{l} \text{execute } C_1 \\ \text{and then} \\ \text{execute } C_2 \end{array} \right.$

A equação semântica em (1) define a representação de um número. A função semântica 'valuation of $_$ ' produz um número baseada em sua definição sintática.

A semântica de um identificador (2) é retornar um número (dado transitório) armazenado na célula *ligada* a este identificador.

Representamos a soma de expressões em (3) pela soma de suas sub-expressões. Ambas são avaliadas produzindo uma *tupla* de dados. A soma é então representada, utilizando os números resultantes.

As equações de (4) a (9) definem os comandos. Em (4) um *binding* é criado mapeando o identificador I a uma célula de memória reservada e inicializada com valor 0.

A representação de uma atribuição em (5) é armazenar o valor resultante da avaliação da expressão E em uma célula de memória referenciada pelo identificador I . A semântica de um comando **skip** (6) é tão somente finalizar com sucesso.

A regra (7) define a estrutura de seleção. A expressão é avaliada produzindo um valor que é repassado pelo combinador then. A ação composta por or determina uma das sub-ações a serem executadas. No caso da expressão resultar em valor 1 o comando C_1 é executado e para o caso de resultar em 0 o comando C_2 é então escolhido.

Em (8) determinamos a estrutura de repetição, onde a ação básica unfolding determina a execução repetida de sua sub-ação. Por fim, em (9), expressamos a seqüenciação de comandos.

Podemos resumir alguns pontos positivos e negativos da abordagem de semântica de ações como segue:

- (+) A semântica de ações propõe uma forma simples de definição de especificações formais. O estilo de escrita, baseado em palavras da língua inglesa, é um importante fator desta simplicidade;
- (+) As especificações neste paradigma são de fácil leitura, se comparadas por exemplo, às especificações em semântica denotacional;
- (+) A modificação de especificações nesta abordagem é simples, uma vez que não há complexidade no formalismo para a definição de especificações;
- (-) A verificação de propriedades dos programas não é simples, uma vez que a notação de ações é baseada em semântica operacional [Gay02];
- (-) Blocos reutilizáveis não podem ser definidos na semântica de ações tradicional;

Baseados neste último ponto, vamos introduzir no capítulo seguinte algumas abordagens que visam a modularização de especificações em semântica de ações.

CAPÍTULO 4

SEMÂNTICA DE AÇÕES BASEADA EM MÓDULOS E COMPONENTES

Neste Capítulo vamos mostrar duas abordagens para escrita de especificações em Semântica de Ações, visando o reuso de descrições já existentes para a criação de novos projetos. Serão apresentadas também as técnicas propostas por ambas abordagens, além de um exemplo de uma linguagem simples baseada em expressões, especificada utilizando as duas propostas.

4.1 Semântica de Ações baseada em Módulos

Em [DM01] é proposta uma forma de compor linguagens de programação pelo reuso, extensão e combinação de módulos em semântica de ações. Os módulos são compostos separadamente, sendo que um novo módulo é criado pela extensão ou combinação de outros já existentes.

4.1.1 Notação e exemplo do uso de módulos

Cada módulo, na definição de uma linguagem, é composto de duas partes: sintaxe e semântica. A primeira parte define a estrutura (sintaxe abstrata) da linguagem, enquanto que a segunda traz o significado da mesma. A sequência de exemplos a seguir, baseada em [DM01], mostra o uso deste formalismo para a criação de módulos para representar a semântica de expressões:

```
Expressions {  
  syntax:  
    Expr.  
  semantics:  
    datum >= expressible.  
    (*) evaluate: Expr -> action [giving an expressible]  
}
```

O módulo `Expressions` é criado contendo um *sort* sintático `Expr`. Na seção de semântica o *sort* `datum` determina coleções de dados que podem ser utilizadas nas especificações. O operador `">="` introduz o *sort* `expressible` para ser utilizados nas expressões, definindo que este é *sub-sort* de `datum`.

A função semântica `evaluate` é definida como sendo um mapeamento da entidade sintática `Expr` introduzida para uma ação que produz um `expressible`. O módulo anterior serve de base para as seguintes definições:

```
Arithmetic Expressions {
    import Expressions.
syntax:
    Expr ::= Num | [[ Expr "+" Expr ]].
    Num ::= [[ digit+ ]].
    E1, E2 : Expr; N: Num
semantics:
    expressible >= natural.
    (1) evaluate N = give decimal string-of-characters N.
    (2) evaluate [[ E1 "+" E2 ]] =
        ( evaluate E1 and evaluate E2) then
            give the sum of (the given natural #1, the given natural#2).
}
```

Neste segundo módulo as definições de `Expressions` são incorporadas pelo uso da palavra chave `import` que precede o nome do módulo em questão. O novo módulo, `Arithmetic Expressions` é então definido por suas seções de sintaxe e semântica. Na primeira seção determinamos que uma expressão pode ser um numeral `Num` ou a soma entre expressões (`Expr "+" Expr`).

Na parte semântica deste módulo definimos pelo uso do operador "`>=`", que `natural` é sub-sort de `expressible`, previamente definido em `Expressions`. Indicamos também duas equações semânticas: em (1) a entidade sintática representada por `N` está mapeada para um numeral; em (2) definimos que as sub-expressões `E1` e `E2` resultam, quando avaliadas, nos números naturais `n1` e `n2`, e a semântica de (`[[ E1 + E2 ]]`) nada mais é que a soma dos numerais indicados. Da mesma maneira que definimos o módulo de `Arithmetic Expressions` podemos definir outros como `Boolean Expressions` e `Relational Expressions` (não apresentados aqui) para especificar expressões booleanas e relacionais, respectivamente.

Podemos então compor uma linguagem de expressões pelo uso de módulos já definidos:

```
Expression Language {
    import Arithmetic Expressions,
        Boolean Expressions, Relational Expressions.
}
```

De maneira similar, é possível descrever uma linguagem de programação completa pela composição das definições de diversos módulos:

```
LING Language {
    import Expression Language, Command Language, Declaration Language
}
```

A definição acima poderia corresponder a descrição de uma linguagem denominada LING, que é composta dos módulos Expression Language, Command Language e Declaration Language. Esta definição, no entanto, é apenas ilustrativa e não serão detalhados os módulos correspondentes. É importante ressaltar que trazemos aqui um exemplo simples utilizando o que foi proposto em [DM01], e logo os detalhes para a descrição de uma linguagem de programação mais complexa não estão enfocados. A combinação de módulos é uma operação muito delicada e para ser efetuada devem ser observadas implicações importantes detalhadas em [DM01].

4.2 Semântica de Ações baseada em Componentes

Para introduzir os *componentes* de especificação, Moura e Menezes propuseram em [MM01] a Semântica de Ações Baseada em Componentes (*Component-Based Action Semantics*), que é uma extensão da Semântica de Ações definida por [Mos92]. Duas notações foram introduzidas: a notação de componentes; e a notação das linguagens de programação. Trazemos aqui uma introdução breve desta proposta, que pode ser vista de forma mais aprofundada no trabalho citado.

4.2.1 Notação de Componentes e Linguagens de Programação

Os componentes, assim como os módulos de [DM01], são constituídos de duas partes básicas: sintaxe e semântica. A notação de componentes permite que estruturas reutilizáveis sejam definidas, agrupando estas duas partes. A notação para linguagens de programação dá subsídio para a criação, uso e extensão da especificação de uma linguagem de programação.

A seqüência de exemplos a seguir está baseada em [MM01]. Inicialmente vamos mostrar um componente genérico denominado *binary* utilizado sobre expressões. Três parâmetros são esperados: o tipo da expressão, a string que corresponde ao operador e o *yielder* que representa o resultado a ser retornado.

```
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 #3
  then
    | give y
```

O componente anterior serve de esqueleto para a criação de outros componentes. Por exemplo, podemos definir a soma, subtração, multiplicação e divisão de expressões da seguinte forma:

```
Expression = ... ++ binary Expression "+" (the sum of them) ++  
...
```

O operador "++" representa a composição de componentes. A aplicação deste operador, no caso acima, permite definir o componente `Expression` como resultado da composição dos componentes relativos à definição dos operadores da linguagem.

Entretanto, a notação de componentes não nos permite representar por completo uma linguagem [MM01]. Desta forma, uma notação específica para linguagens de programação se faz necessária, possibilitando a criação, composição e extensão especificações. A notação de linguagens se baseia no uso de componentes previamente definidos para a obtenção da descrição da linguagem. Como exemplo, vamos supor que precisamos definir uma linguagem de expressões. Vamos utilizar o componente `binary` visto acima, de forma a obter o componente que descreve a semântica de expressões aritméticas:

```
extends [ _ ] with _ :: identifier, component -> language
```

```
Expressions =  
| extends [ Expression ] with  
| | binary E "+" (sum of them) ++  
| | binary E "-" (difference of them) ++  
| | binary E "*" (product of them) ++  
| | binary E "/" (division of them) ++  
| ...
```

A definição de `Expressions` representada acima utiliza o operador `extends` que tem a função de estender um componente com nova especificação. Neste caso, como o componente não existe, este operador tem a função de criá-lo, considerando as diversas utilizações de `binary` para os quatro tipos de expressão (soma, subtração, multiplicação e divisão) compostas pelo operador "++". O resultado final de `extends` é então uma linguagem.

Nesta contexto, uma linguagem mais complexa pode ser criada utilizando esta linguagem de expressões. A composição pode ser efetuada com o uso do operador `and` indicado a seguir:

```
_ and _ :: language description, language description -> language  
description
```

```

LING =
| Declaration and Command and Expression
and
| extends [ Program ] with
| syntax [[ "declare" Declaration "in" Command ]]
| semantics
| | | semantics of the subcomponent #1
| | hence
| | | semantics of the subcomponent #2

```

A definição anterior poderia especificar a mesma linguagem LING, apresentada na Seção 4.1.1. Esta linguagem é criada pela composição de Declaration, Command e Expression. A representação detalhada de uma linguagem de programação completa utilizando esta idéia pode ser vista em [MM01].

4.3 Aspectos importantes no uso de Módulos e Componentes

A utilização de módulos ou componentes constitui uma abordagem atual às especificações em Semântica de Ações. Sua utilização é essencial no contexto onde partes de especificações podem produzir novas especificações, economizando tempo e ressaltando a clareza da descrição. Apresentamos a seguir alguns pontos (positivos e negativos) que ao nosso entendimento são relevantes na utilização de módulos e componentes:

Módulos

- (+) A notação para definição dos módulos é de simples aplicação. O resultado final apresenta fácil legibilidade.
- (+) Para fins de reuso a técnica se apresentou bem eficiente. Partes de especificação descritas em outros módulos podem ser agregadas a novas especificações.
- (-) O corpo do módulo importado é inteiramente incluído no modulo que o está importando [DM01].
- (-) A extensão de módulos consiste em agregar um módulo na especificação atual para gerar uma nova versão. Isto implica na repetição da especificação a cada versão especializada. Por exemplo, no caso de um módulo Expression de expressões, as definições deste módulo precisam ser incorporadas nas versões de ArithExpression, RelationalExpression, etc.
- (+) Existe ferramental para a combinação de módulos, pela inclusão de outros na especificação atual.

- (-) A combinação de módulos pode gerar inconsistência. Definições conflitantes (sorts, equações semânticas distintas para a mesma regra sintática) podem tornar a especificação inconsistente, sendo necessária a utilização de um ferramental específico para a correção deste problema, conforme apresentado em [DM01].

Componentes

- (+) A notação de componentes e linguagens de programação provê um ferramental bastante avançado para a criação de componentes e especificação de linguagens.
- (-) As funções e operadores disponibilizados são difíceis de serem utilizados. A definição dos componentes é complexa e sua leitura é difícil. Não fica explícita a semântica das partes.
- (+) Os componentes disponibilizados são no geral genéricos. O reuso é bem empregado pois não se fundamenta na repetição da parte de especificação, mas sim no uso de um componente na nova especificação.
- (+) A extensão é automática uma vez que os componentes são definidos para serem genéricos, e para podem ser especializados na nova especificação.
- (+) A combinação dos componentes é efetuada pela utilização de funções e operadores. As linguagens são construídas da mesma maneira, havendo ainda a possibilidade de uso de uma biblioteca de componentes [MM01].

CAPÍTULO 5

SEMÂNTICA DE AÇÕES ORIENTADA A OBJETOS

Neste capítulo vamos propomos uma extensão da Semântica de Ações a qual denominamos Semântica de Ações Orientada a Objetos. Primeiramente, apresentaremos nossa proposta de maneira informal com base em exemplos, seguida da sintaxe da notação de ações para este novo paradigma. Por fim, apresentaremos a semântica da notação elaborada.

5.1 Introdução

Algumas vezes verificamos que certas partes de uma especificação são muito similares a outras já existentes. Isto faz com que sejamos motivados a reutilizar estas partes. Na maioria dos casos alguns ajustes são necessários para adequá-las a uma nova especificação, todavia é possível constatar uma estrutura comum. Em Semântica de Ações, organizamos o trabalho em partes, de forma intuitiva; porém não há ferramental para uma real divisão em módulos. Alguns trabalhos têm sido relatados com este fim.

Um estudo foi inicialmente apresentado em [DM01]. Neste relato, partindo de trechos de uma linguagem e uma notação bem definida para especificação em Semântica de Ações incluindo módulos, é possível evidenciar a divisão de módulos e compor uma linguagem a partir destes. Outro estudo importante neste sentido foi efetuado por [MM01], onde se evidenciaram em larga escala aspectos de reuso e adequação de partes de linguagens a novas especificações. Uma breve introdução destes trabalhos foi apresentada no Capítulo 4. Neste contexto, onde a composição de linguagens pode ser efetuada de maneira mais eficiente, encontramos uma possibilidade de expandir ainda mais a funcionalidade dos módulos ou componentes. Essa idéia consiste em tratá-los como objetos, desfrutando das características que estes nos fornecem.

Na Seção 5.1.1 vamos apresentar uma breve introdução sobre Orientação a Objetos. Abordaremos alguns conceitos fundamentais deste paradigma, enfocando principalmente os tópicos aplicados ao nosso trabalho. Na Seção 5.1.2 introduziremos informalmente a utilização de nossa proposta com o uso de exemplos.

5.1.1 Classes e Objetos

Um *objeto* é uma abstração de um conceito bem definido do mundo real. Cada objeto possui características que o distinguem de outros, determinando uma entidade única. Um conjunto de objetos pode apresentar propriedades (estrutura de dados ou atributos) e comportamento (operações) semelhantes, o que nos permite agrupá-los e determinar desta forma uma *classe de objetos*. As classes descrevem um conjunto (possivelmente infinito) de objetos.

Um objeto também pode ser visto como uma *instância* de uma classe. Cada instância (objeto) tem sua própria identidade, possuindo todos os atributos e operações descritos pela classe. Um *atributo* é um valor mantido pelos objetos de uma classe, sendo que estes valores podem diferir para cada instância de objeto. Instâncias diferentes de objetos podem ter valores iguais ou distintos para um dado atributo porém os nomes de atributos devem ser únicos dentro de uma classe.

5.1.1.1 Atributos e Operações dos Objetos

Objetos podem ter um ou mais valores associados. Por exemplo, uma pessoa possui um *nome* e um *endereço*. Estes atributos serão eventualmente modificados por meio de *operações* ou *métodos* sobre o objeto referenciado.

Um *método* é uma função ou transformação aplicada sobre objetos. Os métodos, implicitamente, têm um objeto-alvo (o próprio objeto) e são aplicáveis a todos os objetos de uma classe. Especificamente, uma operação modifica um ou mais valores de atributos que estão contidos no objeto.

5.1.1.2 Generalização e Especialização

Generalização é o relacionamento entre uma classe e uma ou mais versões refinadas desta. Denominamos *superclasse* a classe em processo de refinamento, e *sub-classes* as classes refinadas (especializadas).

A *superclasse*, por definição, disponibiliza características que podem ser utilizadas por suas *sub-classes*. Dizemos que estas *herdam* esses atributos e métodos.

Para uma visão mais aprofundada da abordagem orientada a objetos consultar [Rum94, Mey97].

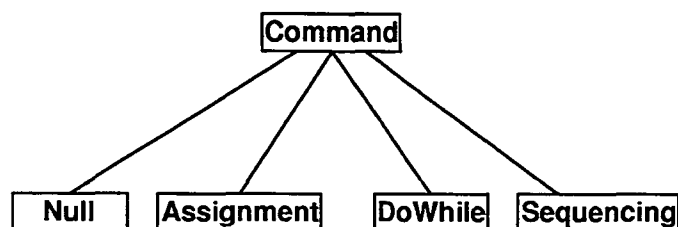


Figura 5.1: Estrutura de objetos da linguagem do Exemplo 5.1.1

5.1.2 Semântica de Ações Orientada a Objetos

Nossa proposta constitui a realização de estudos visando a divisão de especificações em módulos ou componentes em Semântica de Ações, adicionando algumas facilidades advindas da área de Orientação a Objetos.

Em nossa abordagem, uma *hierarquia* de classes é definida em função da estrutura sintática da linguagem. Cada frase (declaração, comando, expressão) da linguagem tem correspondência direta com um ou mais objetos que a representam. Inicialmente vamos mostrar o uso de nossa proposta aplicada a um exemplo simples de uma linguagem de comandos.

Exemplo 5.1.1 (Sintaxe abstrata de comandos)

- Command = `["skip"]` |
`[[Identifier "!=" Expression]` |
`["do" Command "while" Expression]` |
`[[Command ";" Command]` .

O não-terminal Command define os comandos da linguagem do Exemplo 5.1.1. Respectivamente, os comandos são: o comando nulo, a atribuição, o comando de repetição (while) e a seqüenciação.

Os comandos possuem características comuns. Todavia, cada um apresenta um formato e um significado particular. O comando **skip**, que indica a ação nula, por exemplo, difere dos demais tanto na forma de sua escrita como em seu comportamento. Esta situação pode ser representada por meio de uma *hierarquia de classes*, descrita na Figura 5.1.

Command representa as características comuns aos comandos. É por meio desta classe que podemos expressá-las. Suas sub-classes representam as características específicas a cada um dos comandos. São elas: Null, Assignment, DoWhile e Sequencing.

O uso de objetos requer modificações na forma de escrita das especificações em Semântica de Ações. Uma classe em Semântica de Ações Orientada a Objetos deve ser composta de duas partes básicas: sintaxe e semântica, assim como em [DM01] e [MM01].

Exemplo 5.1.2 (Semântica de um Comando)

```
Class Command
  syntax:
    Com
  semantics:
    execute _ : Com -> Action
End Class
```

No Exemplo 5.1.2 definimos a classe `Command` que representa comandos de uma linguagem. Esta é utilizada como base para a posterior definição de diversos comandos.

Na seção de sintaxe (`syntax`) definimos o *sort* sintático `Com`. Este representa a raiz da árvore sintática de comandos.

Uma função semântica '`execute _`' é criada na seção de semântica (`semantics`) indicando que um *sort* `Com` está mapeado para uma ação (`Action`). Em Semântica de Ações Orientada a Objetos, as funções e equações semânticas têm o mesmo papel dos *métodos* em Orientação a Objetos [Rum94, Mey97]. Logo, a função `execute` pode ser vista como um método de `Command`, podendo ser redefinida (*sobrecarregada*) por suas sub-classes.

Exemplo 5.1.3 (Semântica de um Comando Do-While)

```
Class DoWhileCommand
  extending Command
  using E:Expression, C:Command
  syntax:
    Com ::= "do" C "while" E
  semantics:
    execute[[ "do" C "while" E ]]=
      unfolding
        execute C
      and then
        evaluate E
      then
        unfold
      else
        complete
End Class
```

Com base na diretiva `extending` indicamos que a classe `DoWhileCommand` é uma extensão de `Command` (Exemplo 5.1.3). Dois objetos (`E:Expression` e `C:Command`) são *instanciados* pelo uso da diretiva `using` para serem posteriormente utilizados.

Na seção de sintaxe, acrescentamos a descrição da estrutura de um novo comando, redefinindo a árvore sintática. Um método '`execute _`' (equação semântica) é definido para este comando e sua semântica de ações é especificada.

Em nossa abordagem, as ações representadas por *execute* C ou *evaluate* E, por exemplo, são obtidas pela chamada dos métodos *execute* e *evaluate* às instâncias dos objetos *Command* e *Expression* (C e E), respectivamente. Desta maneira, *interfaces* devem ser disponibilizadas pelos objetos para expressar sua semântica.

5.1.3 Implicações iniciais no tratamento de Módulos como Objetos

A abordagem dos componentes como objetos nos permite conhecer e expandir uma série de características. Para usufruir de suas vantagens é necessário verificar algumas regras que disciplinam sua utilização.

Em Semântica de Ações Orientada a Objetos, um objeto é uma instância de uma classe que encapsula sintaxe e semântica. Podemos destacar três preocupações fundamentais na abordagem de módulos como objetos:

Encapsulamento: O procedimento de obtenção dos resultados por parte do objeto deve estar oculto e não visível aos demais;

Interface bem definida: O objeto em Semântica de Ações deve expor interfaces que permitam a troca de mensagens com outros objetos;

Ocultamento: O acesso aos atributos deve estar restrito; a troca de informações está condicionada à disponibilização de interfaces.

Baseados nestas premissas, vamos formalizar o uso de Objetos em Semântica de Ações. Definiremos, nas próximas seções, a sintaxe da notação exemplificada anteriormente, assim como sua semântica.

5.2 Sintaxe da Notação

Uma especificação em Semântica de Ações Orientada a Objetos é um conjunto finito de classes. O relacionamento entre estas é especificado por meio de diretivas, que indicam os objetos instanciados pela classe atual, assim como a hierarquia a qual esta pertence. A estrutura das frases de uma linguagem é dada pela especificação de sua árvore sintática. A semântica, por sua vez, é expressa pela notação de ações [Mos92].

Mapeamos a sintaxe abstrata de uma linguagem para essa hierarquia, seguindo a notação descrita a seguir:

5.2.1 Classes

- (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 ]]`

A declaração de uma Classe (1) é definida pelo token `Class` seguido de seu nome, de um corpo, e de `End-Class`. No corpo da classe (2) indicamos inicialmente sua *classe-base* e os objetos por ela utilizados. As seguintes diretivas podem ser empregadas:

- `extending`: informa que a Classe Atual está estendendo uma Classe base (3), ou seja, formando uma hierarquia. O operador de escopo representado por `::`, permite acessar determinada classe dentro da hierarquia;
- `using`: objetos utilizados (4) internamente na Classe Atual. Cada objeto instanciado é representado por um identificador, seguido de `:` e da classe correspondente (5).

Exemplo 5.2.1 (Uma classe de declaração de Constante)

```
Class ConstantDeclaration
    extending Declaration
    using E:Expression, T:Type
    ...
End Class
```

A classe `ConstantDeclaration` representa a semântica da declaração de constantes de uma linguagem de programação. Esta classe é uma extensão da classe `Declaration`, que corresponde à classe base de quaisquer formas de declaração. Para seu uso interno, a classe exemplo instancia dois objetos: `Expression` e `Type`, representados pelos identificadores `E` e `T`, respectivamente. A definição de classes será detalhada a seguir.

5.2.1.1 Definição das Classes

- (1) Class-Definition =
 [["syntax" ":" Syntactic-Part "semantics" ":" Semantic-Part]]
- (2) Syntactic-Part =
 [[TokenName | TokenName "::=" syntax-tree]]
- (3) Semantic-Part =
 [[{ Semantic-Functions }? { Semantic-Equations }?].
- (4) Semantic-Functions =
 [[Semantic-Function] | [[Semantic-Function Semantic-Functions]].
- (5) Semantic-Function =
 [[Function-Name "_" Tokens "->" "Action" | data]].
- (6) Tokens =
 [[TokenName] | [[TokenName "," Tokens]].
- (7) Semantic-Equations =
 [[Semantic-Equation] | [[Semantic-Equation Semantic-Equations]].
- (8) Semantic-Equation =
 [[Function-Name "[[" syntax-tree "]" "=" Action]].

A definição de uma classe (1) é composta das seções de sintaxe e semântica, representadas como a seguir:

- **syntax:** Nesta parte indicamos a sintaxe abstrata da frase corrente. A parte sintática, descrita em (2), pode ser um nome de token ou o mesmo seguido de uma árvore sintática (*syntax tree*) como em [Mos92] (apêndice E).
- **semantics:** As funções semânticas e equações semânticas, referenciadas aqui como *métodos* sobre os objetos, provêm a semântica das frases. As regras (4) a (6) especificam as funções semânticas. Definimos uma função semântica (5) como sendo um mapeamento de entidades sintáticas (Tokens), para uma ação ou dado (Action ou data). Equações semânticas são definidas pelas regras (7) e (8). Em (8) definimos a notação para as equações. Um nome de função (Function-Name) é associado a determinada árvore-sintática (*syntax-tree*), e sua denotação, por sua vez, é uma ação (Action).

Vejamos a seguir mais detalhadamente o exemplo de declaração de constantes.

Exemplo 5.2.2 (A semântica de uma declaração de Constante)

```
Class Declaration
  using I:Identifier
  syntax:
    Dec
  semantics:
    elaborate _ : Dec -> Action
End Class

Class ConstantDeclaration
  extending Declaration
  using E:Expression, T:Type
  syntax:
    Dec ::= "const" I ":" T "=" E
  semantics:
    elaborate[["const" I ":" T "=" E]] =
      evaluate E then
        bind I to the value
End Class
```

No Exemplo 5.2.2 são definidas duas classes, construindo uma hierarquia de declarações composta por *Declaration* e *ConstantDeclaration*. A primeira, que corresponde à *classe-base*, instancia o objeto *Identifier*, disponibilizado à sua *sub-classe*. O *token* *Dec* é definido na parte sintática e um método (função semântica) ‘*elaborate _*’ é criado na seção correspondente. Este método define um mapeamento de uma frase, representada por *Dec*, para uma ação (*Action*).

A segunda classe, apresentada parcialmente no Exemplo 5.2.1, é uma extensão da primeira e representa uma declaração de constante. Os objetos *Expression* e *Type* são instanciados para serem utilizados internamente. Na seção *syntax*, redefinimos *Dec* para especificar declarações de constantes. O método *elaborate* é sobrecarregado pela definição da equação semântica *elaborate* sobre a nova sintaxe, e sua denotação é indicada na forma de uma ação. A semântica da declaração de constante é efetuar um *binding* entre o identificador (*I*) e o valor fornecido pelo método *evaluate* do objeto *Expression* em questão.

A notação de ações definida neste trabalho mantém a estrutura introduzida por [Mos92] e representa um sub-conjunto da notação original, baseado no trabalho de [Mou93]. A diferença, todavia, está na interpretação das funções definidas pelo usuário. Em Semântica de Ações, a ação correspondente a ‘*evaluate E*’ está determinada por equações semânticas sobre a sintaxe representada por *E*. Em nossa abordagem, *evaluate* é um método sobre um objeto *E*, estruturado exatamente como uma equação semântica, que produz a mesma ação.

5.2.1.2 Notação de Ações

Apresentamos a seguir a sintaxe da notação de ações, que será utilizada para definir a semântica das frases (corpo dos métodos).

(1) Action =

```
[[ "complete" ]] | [[ "fail" ]] | [[ "unfold" ]] |  
[[ Action "and" Action ]] | [[ Action "and then" Action ]] |  
[[ "unfolding" Action ]] | [[ "give" Yielder ]] |  
[[ "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 ]].
```

(2) Yielder =

```
[[ "the" Sort "#"n ]] |  
[[ "the" Sort "bound" "to" k ]] |  
[[ "the" Sort "stored" "in" Yielder ]] |  
[[ Yielder "with" Yielder ]] |  
[[ "closure" Yielder ]] |  
[[ "abstraction of" Action ]].
```

(3) Sort =

```
[[ "bindable" ]] | [[ "cell" ]] | [[ "cell" "[" Sort "]" ]] |  
[[ "storable" ]] | [[ "abstraction" ]] | [[ "datum" ]] |  
[[ "integer" ]] | [[ "value" ]] | [[ "truth-value" ]] |  
[[ Sort "[" Sort "]" ]].
```

As regras de (1) a (3) definem a sintaxe da Notação de Ações e Dados, com base em [Mou93]. Em (1) definimos o conjunto de *ações* a serem utilizadas nas especificações baseadas em nossa abordagem. Os *yielders* e *sorts* estão descritos pelas regras (2) e (3), respectivamente. A semântica de nossa notação está descrita na próxima Seção.

5.3 Semântica da Notação

As classes-base definidas a partir da notação orientada a objetos, *implicitamente*, fazem parte de uma hierarquia. Isto se dá porque em todas as especificações deste paradigma, assumimos a existência uma super-classe, denominada *State*, a partir da qual todas as demais são geradas.

Esta classe possui atributos genéricos correspondentes a dados transitórios, bindings e armazenamento e ao mesmo tempo disponibiliza operações, que permitem interagir com estes

atributos. Estes atributos e operações são visíveis às suas sub-classes, ou seja, por quaisquer classes mais específicas.

A semântica da notação de ações está definida no formato de regras de inferência e está baseada na proposta de [Mou93]. Vamos introduzir a seguir a classe principal das especificações em Semântica de Ações Orientada a Objetos.

5.3.1 A Classe *State*

A Semântica de Ações Orientada a Objetos caracteriza-se pela criação de uma Estrutura de Classes que representam a semântica de uma linguagem. Estas classes fazem uso de noções relativas dos Objetos (Capítulo 4) que permitem principalmente o reuso e a extensão das especificações em Semântica de Ações.

As classes definidas neste paradigma são sub-classes diretas de *State*. Esta, por sua vez, é responsável por representar a Semântica de Ações original [Mos92]. A classe *State* possui como atributos os dados transitórios t , um ambiente b (*bindings*) e um armazenamento s . Os métodos dessa classe correspondem às ações básicas e combinadores em semântica de ações. Definimos o comportamento dos métodos de *State* mediante uma relação da seguinte forma:

$$S = B, t, b, s \vdash o \triangleright o', t', b', s'$$

Sendo B um *ambiente* para definição dos métodos (especificados nas classes elaboradas pelo usuário), t o conjunto de dados transitórios (*transient*), b o conjunto de *bindings* e s o *storage* corrente, dizemos que a operação (*ação*) o , quando executada produz o resultado o' , os dados transitórios t' , os *bindings* b' e modifica o *storage* para s' . Como em [Mou93], o resultado de uma ação pode ser: *completed*, *diverge* ou *fail*¹.

5.3.2 Atributos da classe *State*

Consideramos atributos da classe *State*, basicamente, três tipos de informações distintas representadas pela tripla (t, b, s) , que correspondem aos *transients* (dados transitórios), *bindings* e *storage* atual. Em Semântica de Ações Orientada a Objetos, uma série de operações são disponibilizadas sobre *State*, permitindo a modificação destes atributos.

¹Desconsideramos *escape* por não estar no escopo de nosso trabalho.

5.3.2.1 Operações sobre a classe *State*

Operações são funções sobre os atributos de *State*. Estas permitem mapear a tripla (t, b, s) para (t', b', s') . As operações seguem a classificação de ações e são divididas em:

- *básicas*: utilizadas para controle de fluxo;
- *funcionais*: quando aplicadas aos *transients*;
- *imperativas*: quando aplicadas ao *storage*;
- *declarativas*: quando incidem sobre os *bindings*;
- *reflexivas*: definem abstrações;
- *híbridas*: apresentam *comportamento múltiplo*, ou seja, repercutem em mais de um tipo de atributos.

O ambiente para definição de métodos B somente é lido pelas operações anteriores. A modificação de B será comentada na última seção deste capítulo.

É importante ressaltarmos que as operações em Semântica de Ações Orientada a Objetos, correspondem exatamente às *ações* da Semântica de Ações proposta em [Mos92]. Desta forma utilizaremos ambos os termos como sinônimos deste ponto em diante.

Nas próximas seções, introduziremos a semântica formal da notação de ações orientada a objetos, intercalando as regras com exemplos da utilização das operações que estão sendo definidas.

5.3.3 Operações Básicas

Estas operações são aplicadas ao controle de fluxo. Elas apresentam comportamento igual aos elementos da faceta básica descritos em semântica de ações. As regras (5.1) a (5.11) expressam o significado das operações básicas.

$$B, t, b, s \vdash \text{complete} \triangleright \text{completed}, \{\}, \{\}, s \quad (5.1)$$

$$B, t, b, s \vdash \text{fail} \triangleright \text{failed}, \{\}, \{\}, s \quad (5.2)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t, b, s_1 \vdash a_2 \triangleright \text{completed}, t_2, b_2, s_2}{\text{mergeable } t_1 t_2 \quad \text{mergeable } b_1 b_2} \quad (5.3)$$

$$B, t, b, s \vdash a_1 \text{ and } a_2 \triangleright \text{completed}, t_1 \oplus t_2, b_1 \oplus b_2, s_2$$

$$\frac{B, t, b, s \vdash a_1 \triangleright o_1, t_1, b_1, s_1 \quad o_1 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ and } a_2 \triangleright o_1, t_1, b_1, s_1} \quad (5.4)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t, b, s_1 \vdash a_2 \triangleright o_2, t_2, b_2, s_2 \quad o_2 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ and } a_2 \triangleright o_2, t_2, b_2, s_2} \quad (5.5)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad \text{mergeable } t_1 t_2 \quad B, t, b, s_1 \vdash a_2 \triangleright \text{completed}, t_2, b_2, s_2 \quad \text{mergeable } b_1 b_2}{B, t, b, s \vdash a_1 \text{ and then } a_2 \triangleright \text{completed}, t_1 \oplus t_2, b_1 \oplus b_2, s_2} \quad (5.6)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright o_1, t_1, b_1, s_1 \quad o_1 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ and then } a_2 \triangleright o_1, t_1, b_1, s_1} \quad (5.7)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t, b, s_1 \vdash a_2 \triangleright o_2, t_2, b_2, s_2 \quad o_2 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ and then } a_2 \triangleright o_2, t_2, b_2, s_2} \quad (5.8)$$

$$\frac{B, t, b, s \vdash a \text{ [unfolding } a/\text{unfold}] \triangleright o', t', b', s'}{B, t, b, s \vdash \text{unfolding } a \triangleright o', t', b', s'} \quad (5.9)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright o_1, t_1, b_1, s_1 \quad o_1 \neq \text{failed}}{B, t, b, s \vdash a_1 \text{ or } a_2 \triangleright o_1, t_1, b_1, s_1} \quad (5.10)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{failed}, \{\}, \{\}, s_1 \quad B, t, b, s_1 \vdash a_2 \triangleright o_2, t_2, b_2, s_2}{B, t, b, s \vdash a_1 \text{ or } a_2 \triangleright o_2, t_2, b_2, s_2} \quad (5.11)$$

Em (5.1) expressamos a semântica da operação **complete**. Esta, quando aplicada aos atributos t, b, s , resulta em **completed**, não produz dados transitórios e *bindings*, além de não modificar o *store* atual. A operação **fail**, expressada na regra (5.2), produz o resultado **failed** e tem o mesmo comportamento de (5.1) sobre os atributos.

Exemplo 5.3.1 (Uso de operação básica na especificação de um comando nulo)

```

Class NullCommand
  extending Command
  syntax:
    Com ::= "skip"
  semantics:
    execute[["skip"]] =
      complete
End Class

```

O significado de um comando **skip** corresponde a uma operação **complete** sobre os atributos implícitos (t, b, s) herdados pela classe que o representa. O resultado é **completed** e os atributos são modificados de acordo com o comportamento mencionado.

As regras (5.3) a (5.5) definem o comportamento da operação *and*. Em (5.3) determinamos o resultado de ' A_1 and A_2 ' quando ambas as sub-ações resultam em *completed*. Os dados transitórios e *bindings* de entrada são repassados a ambas as ações. O *store* de entrada da ação A_2 , todavia, é aquele resultante da execução de A_1 (s_1). O predicado *mergeable* é utilizado para garantir que os conjuntos não tenham os mesmos elementos. A operação ($\oplus$) produz a união os conjuntos indicados, conforme descrito em [Mou93]. Em (5.4), definimos que o resultado final no caso da primeira sub-ação não finalizar com sucesso, corresponde às modificações efetuadas pela mesma sobre os atributos. De maneira similar, caso a primeira ação finalize com sucesso, conforme (5.5), determinamos que o resultado final, caso a segunda sub-ação falhe ou execute ininterruptamente, corresponde às alterações produzidas por essa segunda ação.

Para fins de simplificação adotamos o comportamento acima descrito para a operação *and* de acordo com a semântica do *combinador and* definida por [Mou93]. Esta abordagem difere do que foi proposto originalmente por [Mos92], onde as sub-ações de ' A_1 and A_2 ' são intercaladas. A operação *and then*, descrita pelas regras de (5.6) à (5.8) tem o mesmo comportamento de *and*, supracitado. Este fato não representa uma restrição muito significativa. Caso exista a necessidade de especificar o comportamento com intercalação (*interleaving*), podemos utilizar as técnicas descritas em [Mus96, Mos99].

Exemplo 5.3.2 (Operação básica "and then" na seqüenciação de comandos)

```

Class SequentialCommand
  extending Command
  using C1:Command, C2:Command
  syntax:
    Com ::= C1 ";" C2
  semantics:
    execute[[C1 ";" C2]]=
      execute C1 and then execute C2
End Class

```

Os comandos acima são seqüenciados. Na especificação desta seqüência, as mudanças efetuadas no *store* são repassadas como entrada para a execução do segundo comando.

A operação *unfolding*, definida pela regra (5.9), expressa um comportamento recursivo. As modificações finais de '*unfolding A*' são expressas pelos valores dos atributos após a última execução de A . A aplicação desta operação pode ser verificada na especificação do comando "Do-While", visto no Exemplo 5.1.3, e transcrito a seguir.

Exemplo 5.3.3 (Semântica de um Comando Do-While)

```
Class DoWhileCommand
...
  semantics:
    execute[[ "do" C "while" E ]]=
      unfolding
        execute C
      and then
        evaluate E
      then
        unfold
      else
        complete
End Class
```

No Exemplo 5.3.3, a ação associada a *unfolding* está especificada no texto *sob* a definição da operação em questão. A semântica do objeto *C* é fornecida por seu método *execute*. O método *evaluate* expressa a semântica da expressão representada pelo objeto *E*. Caso o resultado seja verdadeiro, o processo recomeça; sendo este comportamento determinado por *unfold*. Caso contrário, o comando finaliza com término normal.

O comportamento da operação (combinador) *or* é determinado pelas regras (5.10) e (5.11). Se a execução da primeira sub-ação completa ou diverge, o resultado final desta operação será aquele produzido por esta ação, conforme a regra (5.10). Em (5.11), definimos que, caso a primeira regra falhe, as mudanças de *store* são repassadas à segunda sub-ação que produz o resultado final.

Exemplo 5.3.4 (Semântica da avaliação de um Identificador)

```
Class IdentifierExpression
  extending Expression
  using I:Identifier
  syntax:
    Exp ::= I
  semantics:
    evaluate [[ I ]] =
      give the value bound to I
    or
      give the value stored in the cell
      bound to I
End Class
```

Em 5.3.4, a operação *or* é utilizada para definir a escolha entre duas sub-operações. Estas definem a avaliação de um identificador que pode ser uma constante no primeiro caso ou variável no segundo. O resultado final dos atributos está condicionado à sub-operação escolhida.

Um tipo especial de operação básica, denominada *operação produtora*, é definida em (5.12). Esta tem a função de retornar as informações dos atributos (t, b, s) , durante a execução de uma operação, não havendo nenhuma mudança nos valores dos mesmos. Em Semântica de Ações este comportamento é expresso pelos *yielders* [Mos92] e as regras que definem o comportamento dos *yielders* têm a forma:

$$B, t, b, s \vdash y \triangleright d \quad (5.12)$$

As seguintes operações produtoras podem ser exemplificadas de acordo com a seguinte configuração dos atributos (t, b, s) : $\{0 \mapsto 3\}$, $\{x \mapsto \text{true}, y \mapsto \text{cell0}\}$, $\{\text{cell0} \mapsto 3\}$;

‘the integer #0’ retorna um dado transitório simples 3;

‘the truth-value bound to “x”’ retorna o bindable *true*;

‘the integer store in the cell bound to “y”’ retorna o *storable* armazenado na célula ligada a “y”, ou seja o valor 3.

5.3.4 Operações Funcionais

São as operações que atuam sobre o atributo de dados transitórios dos objetos. São representadas pelas regras de (5.13) a (5.19).

$$\frac{B, t, b, s \vdash y \triangleright d}{B, t, b, s \vdash \text{give } y \triangleright \text{completed}, \{0 \mapsto d\}, \{\}, s} \quad (5.13)$$

$$\frac{B, t, b, s \vdash y \triangleright \text{true}}{B, t, b, s \vdash \text{check } y \triangleright \text{completed}, \{\}, \{\}, s} \quad (5.14)$$

$$\frac{B, t, b, s \vdash y \triangleright \text{false}}{B, t, b, s \vdash \text{check } y \triangleright \text{failed}, \{\}, \{\}, s} \quad (5.15)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t_1, b, s_1 \vdash a_2 \triangleright \text{completed}, t_2, b_2, s_2 \quad \text{mergeable } b_1 b_2}{B, t, b, s \vdash a_1 \text{ then } a_2 \triangleright \text{completed}, t_2, b_1 \oplus b_2, s_2} \quad (5.16)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright o_1, t_1, b_1, s_1 \quad o_1 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ then } a_2 \triangleright o_1, t_1, b_1, s_1} \quad (5.17)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t_1, b, s_1 \vdash a_2 \triangleright o_2, t_2, b_2, s_2 \quad o_2 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ then } a_2 \triangleright o_2, t_2, b_2, s_2} \quad (5.18)$$

$$\frac{d : r}{B, (n \mapsto d) \cdot t, b, s \vdash \text{the } r \# n \triangleright d} \quad (5.19)$$

Em (5.13) definimos a operação que corresponde à ação fundamental da faceta funcional em Semântica de Ações. O resultado de “give y” é finalizar com sucesso (*completed*), sendo que a tripla (t, b, s) resulta em $(\{0 \mapsto d\}, \{\}, s)$, onde: $\{0 \mapsto d\}$ indica que haverá um único mapeamento de dados transitórios representando o resultado d retornado pela operação produtora y ; $\{\}$ determina que não há *bindings*; e s indica que a memória corrente não é modificada.

Exemplo 5.3.5 (Uso da notação funcional em expressões)

```

Class True-Expression
  extending Expression
  syntax:
    Exp ::= "true"
  semantics:
    evaluate[[ "true" ]] =
      give true
End Class

```

A classe *True-Expression* representa a semântica de uma entidade sintática “true”. O resultado da operação “give true” é um dado transitório $\{0 \mapsto \text{true}\}$.

As regras (5.14) e (5.15) determinam o comportamento da operação “check y”. Dada uma operação produtora y , que resulta em *true* ou *false* dependendo da regra escolhida, o resultado da operação principal no primeiro caso será *completed* gerando para (t, b, s) respectivamente: $\{\}$ indicando que não serão produzidos dados transitórios; $\{\}$ indicando que não há *bindings*; e s indicando que a memória permanece sem modificações. No segundo caso, *check* resulta em *failed* e os atributos (t, b, s) têm resultado igual ao descrito anteriormente.

A operação *then* é especificada pelas regras de (5.16) a (5.18). Em (5.16) definimos o comportamento de $A_1 \text{ then } A_2$ quando ambas as sub-ações finalizam com sucesso (*completed*). A tripla (t, b, s) é repassada a A_1 que produz (t_1, b_1, s_1) . Nesta operação, os *bindings* de entrada são repassados a ambas sub-ações, todavia os dados transitórios e memória são seqüenciados. Desta forma, a sub-ação A_2 recebe como entrada a tripla (t_1, b, s_1) resultando em (t_2, b_2, s_2) . O resultado final da operação *then* são os dados transitórios e memória da segunda sub-ação, sendo

os *bindings*, no entanto, mesclados pela operação $(\oplus)$ conforme [Mou93]. Logo a tripla final é $(t_2, b_1 \oplus b_2, s_2)$.

A regra (5.17) determina que, caso da sub-operação A_1 não finalize com sucesso, o resultado final da tripla será o resultado determinado pela sub-operação, ou seja, (t_1, b_1, s_1) . Em (5.18) definimos o comportamento quando a sub-ação A_1 completa e A_2 não. O resultado final da tripla é determinado por A_2 e resulta em (t_2, b_2, s_2) . Neste caso especial, diferente do término normal representado por (5.17), os *bindings* não são mesclados.

A operação produtora “the” é expressa em (5.19). Dada a tripla (t, b, s) , onde $\{n \mapsto d\} \in t$, o resultado da operação “the $r\#n$ ” é o dado d do tipo r , identificado pelo label $\#n$.

Exemplo 5.3.6 (Semântica de um Comando If-Then-Else)

```

Class IfThenElseCommand
  extending Command
  using E:Expression, C1:Command, C2:Command
  syntax:
    Com ::= "if" E "then" C1 "else" C2
  semantics:
    execute[[ "if" E "then" C1 "else" C2 ]]=
      evaluate E
      then
        check(the truth-value is true) and then
        execute C1
      or
        check(the truth-value is false) and then
        execute C2
End Class

```

A operação principal no Exemplo 5.3.6 é *then*. Os *bindings* da operação, seguindo o comportamento já visto, são distribuídos às sub-ações. Os dados transitórios e memória resultantes da avaliação da expressão (*evaluate E*) são transferidos à segunda sub-ação. Nesta, a operação *or* determina qual dos comandos será executado: das duas possibilidade de *check*, uma irá finalizar com término normal, e esta determinará quais os comandos entre $C1$ e $C2$ serão executados.

As operação produtora “the truth-value is d ”, sendo d correspondente a *true* ou *false*, resulta em “true” ou “false”, dependendo do conjunto de dados transitórios que a ação recebe de *evaluate E*.

5.3.5 Operações Declarativas

As operações declarativas interagem com os *bindings* nos objetos. São determinadas pelas regras de 5.20 a 5.32.

$$\frac{B, t, b, s \vdash y \triangleright d \quad d : \text{bindable}}{B, t, b, s \vdash \text{bind } k \text{ to } y \triangleright \text{completed}, \{\}, \{k \mapsto d\}, s} \quad (5.20)$$

$$\frac{B, t, b, s \vdash a \triangleright \text{completed}, t', b', s'}{B, t, b, s \vdash \text{furthermore } a \triangleright \text{completed}, t', b' \odot b, s'} \quad (5.21)$$

$$\frac{B, t, b, s \vdash a \triangleright o', t', b', s' \quad o' \neq \text{completed}}{B, t, b, s \vdash \text{furthermore } a \triangleright o', t', b', s'} \quad (5.22)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t, b_1, s_1 \vdash a_2 \triangleright \text{completed}, t_2, b_2, s_2 \quad \text{mergeable } t_1 t_2}{B, t, b, s \vdash a_1 \text{ hence } a_2 \triangleright \text{completed}, t_1 \oplus t_2, b_2, s_2} \quad (5.23)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright o_1, t_1, b_1, s_1 \quad o_1 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ hence } a_2 \triangleright o_1, t_1, b_1, s_1} \quad (5.24)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t, b_1, s_1 \vdash a_2 \triangleright o_2, t_2, b_2, s_2 \quad o_2 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ hence } a_2 \triangleright o_2, t_2, b_2, s_2} \quad (5.25)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t, b, s_1 \vdash a_2 \triangleright \text{completed}, t_2, b_2, s_2 \quad \text{mergeable } t_1 t_2}{B, t, b, s \vdash a_1 \text{ moreover } a_2 \triangleright \text{completed}, t_1 \oplus t_2, b_2 \odot b_1, s_2} \quad (5.26)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright o_1, t_1, b_1, s_1 \quad o_1 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ moreover } a_2 \triangleright o_1, t_1, b_1, s_1} \quad (5.27)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t, b, s_1 \vdash a_2 \triangleright o_2, t_2, b_2, s_2 \quad o_2 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ moreover } a_2 \triangleright o_2, t_2, b_2, s_2} \quad (5.28)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t, b_1 \odot b, s_1 \vdash a_2 \triangleright \text{completed}, t_2, b_2, s_2 \quad \text{mergeable } t_1 t_2}{B, t, b, s \vdash a_1 \text{ before } a_2 \triangleright \text{completed}, t_1 \oplus t_2, b_2 \odot b_1, s_2} \quad (5.29)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright o_1, t_1, b_1, s_1 \quad o_1 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ before } a_2 \triangleright o_1, t_1, b_1, s_1} \quad (5.30)$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad t, b_1 \odot b, s_1 \vdash a_2 \triangleright o_2, t_2, b_2, s_2 \quad o_2 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ before } a_2 \triangleright o_2, t_2, b_2, s_2} \quad (5.31)$$

$$\frac{d : \tau}{B, t, (k \mapsto d) \cdot b, s \vdash \text{the } \tau \text{ bound to } k \triangleright d} \quad (5.32)$$

A operação declarativa mais simples está descrita na regra (5.20). “bind k to y ” produz um novo ambiente que contém um único *binding* do identificador k para um valor, resultante da operação produtora y . Com base na tripla (t, b, s) o resultado desta operação é $(\{\}, \{k \mapsto d\}, s)$ onde: $\{\}$ indica que não são produzidos dados transitórios; $\{k \mapsto d\}$ indica o ambiente produzido, onde d é obtido através da operação y ; e s indica que a memória permanece sem alterações.

Exemplo 5.3.7 (Uso da notação declarativa)

```

Class ConstantDeclaration
  extending Declaration
  using E:Expression, T:Type
  syntax:
    Dec ::= "const" I ":" T "=" E
  semantics:
    elaborate[["const" I ":" T "=" E]] =
      evaluate E then
        bind I to the value
End Class

```

A classe anterior representa a declaração de constantes. A operação `then` estabelece a transferência do dado transitório proveniente de `evaluate E` para a segunda sub-ação. Esta, por sua vez, define a criação de um novo ambiente que contém a ligação do identificador I ao valor referenciado por `the value`.

Em (5.21) e (5.22) definimos o comportamento de “furthermore A ”. A tripla (t, b, s) fornecida a esta operação é repassada à sub-operação A . Em (5.21) definimos que o resultado final da operação `furthermore` em caso de sucesso é determinado por $(t', b' \oslash b, s')$, indicando que: os dados transitórios e memória finais são aqueles produzidos pela sub-operação; os *bindings* são obtidos pela operação de *overlay* ($\oslash$), que indica que os mapeamentos verificados no ambiente gerado por A sobrepõem os recebidos por “furthermore A ” em caso de coincidência. A regra (5.22) determina que o resultado final, no caso da ação A não resultar em término normal, é aquele produzido por A , ou seja, (t', b', s') .

A operação “ A_1 hence A_2 ”, descrita de 5.23 a 5.25, permite a passagem dos *bindings* entre sub-operações. A tripla (t, b, s) é repassada a sub-operação A_1 que produz (t_1, b_1, s_1) . Em (5.23) determinamos que, caso a A_1 tenha término normal, seus *bindings* e memória serão repassados como entrada para A_2 . Se a segunda sub-ação tiver término normal, o resultado final de `hence`

será a tripla $(t_1 \oplus t_2, b_2, s_2)$ indicando respectivamente que: os dados transitórios finais são *mesclados* com o uso da operação $(\oplus)$; os bindings e memória finais são aqueles produzidos pela sub-operação A_2 . Caso a sub-operação A_1 não tenha término normal, conforme a regra (5.24), o resultado final será definido apenas por esta sub-operação, que corresponde a (t_1, b_1, s_1) . Na situação em que a sub-operação A_1 tenha término normal, mas A_2 não tenha, o resultado final será fornecido pela segunda operação, representado pela tripla (t_2, b_2, s_2) . Este comportamento está definido em (5.25).

As regras (5.26) a (5.28) definem o comportamento da operação “ A_1 moreover A_2 ”. Esta operação estabelece que os *bindings* produzidos por A_2 sobrepõem os de A_1 . Os dados transitórios e bindings de entrada são repassados a ambas sub-operações. A regra (5.26) determina o comportamento quando as sub-operações A_1 e A_2 têm término normal. A tripla (t, b, s) é repassada à primeira sub-operação que produz (t_1, b_1, s_1) . Apenas a memória é repassada a A_2 , que recebe a tripla (t, b, s_1) resultando em (t_2, b_2, s_2) . O resultado final da operação moreover é $(t_1 \oplus t_2, b_2 \oslash b_1, s_2)$, onde: os dados transitórios são mesclados com a utilização de $(\oplus)$; os bindings da segunda sub-operação sobrepõem os da primeira com o uso de $(\oslash)$; a memória final é fornecida pela segunda sub-operação. O comportamento caso não haja término normal na sub-operação A_1 está definido por (5.27). O resultado final da operação, com base na entrada (t, b, s) é aquele determinado por A_1 , ou seja, (t_1, b_1, s_1) . Caso a segunda sub-operação não tenha término normal, conforme (5.28), o resultado final é definido por esta, ou seja, (t_2, b_2, s_2) .

Exemplo 5.3.8 (Exemplo de declaração local a um comando)

```

Class DeclarationCommand
  extending Command
  using D:Declaration
  syntax:
    Com ::= "declare" D "used-in" C
  semantics:
    execute[["declare" D "used-in" C]] =
      furthermore elaborate D
    hence
      execute C
End Class

```

A operação “ A_1 before A_2 ” está descrita pelas regras (5.29) a (5.31). Seu comportamento é análogo a “furthermore A_1 hence A_2 ”.

A operação produtora “the r bound to k ”, definida em (5.31), permite a localização de determinado binding. Seu objetivo é retornar o valor representado por r que está ligado ao identificador k .

5.3.6 Operações Imperativas

As operações para armazenamento de valores, liberação de células e recuperação de informações em memória são determinadas pelas regras 5.33 a 5.35, a seguir:

$$\frac{B, t, b, s \vdash y_1 \triangleright d \quad B, t, b, s \vdash y_2 \triangleright c \quad s = (c \mapsto d_0) \cdot s' \quad c : \text{cell } [r] \quad d : r}{B, t, b, s \vdash \text{store } y_1 \text{ in } y_2 \triangleright \text{completed}, \{\}, \{\}, \{c \mapsto d\} \odot s} \quad (5.33)$$

$$\frac{B, t, b, s \vdash y \triangleright c \quad s = (c \mapsto d) \cdot s'}{B, t, b, s \vdash \text{deallocate } y \triangleright \text{completed}, \{\}, \{\}, s} \quad (5.34)$$

$$\frac{B, t, b, s \vdash y \triangleright c \quad s = (c \mapsto d) \cdot s' \quad d : r}{B, t, b, s \vdash \text{the } r \text{ stored in } y \triangleright d} \quad (5.35)$$

A regra (5.33), define o comportamento de “store y_1 in y_2 ”. Dada a tripla (t, b, s) , o resultado final desta operação é descrito por $(\{\}, \{\}, \{c \mapsto d\} \odot s)$, indicando que: não serão produzidos dados transitórios ou bindings, mas o store s será atualizado com este novo mapeamento. Os pré-requisitos desta operação são que: as operações produtoras y_1 e y_2 devem resultar respectivamente em um dado d e uma célula c ; a célula não deve estar alocada; e o tipo do dado a ser armazenado na célula, identificado por r , deve corresponder ao tipo do dado d efetivamente.

Exemplo 5.3.9 (Uso da notação imperativa em um Comando de atribuição)

```

Class AssignmentCommand
  extending Command
  using I:Identifier, E:Expression
  syntax:
    Com ::= I ":=" E
  semantics:
    execute[[I ":=" E]]=
      evaluate E
    then
      store the value in the cell bound to I
End Class

```

No exemplo (5.3.9), um valor resultante da avaliação da expressão E é repassado pela operação `then` à segunda sub-operação. Esta, por sua vez, armazena o valor na célula relacionada ao identificador em questão, modificando assim a memória atual.

A operação “`deallocate  $y$` ” propõe a liberação de células de memória, conforme a regra (5.34). Dada a tripla (t, b, s) , esta operação retorna $(\{\}, \{\}, s')$ indicando que não são produzidos dados transitórios ou bindings, mas apenas a memória resultante da desalocação, identificada por s' .

O *yielder* identificado por y deve produzir uma célula c , que esteja disponível no store atual indicado por s .

A operação produtora “the r stored in y ” retorna um valor do tipo r que está armazenado na célula representada por y . Esta operação aparece no exemplo (5.3.4), relacionada à avaliação de um identificador em expressões.

5.3.7 Operações Reflexivas

Estas operações são utilizadas para especificar *abstrações*. Em Semântica de ações, uma abstração é um tipo de dado que incorpora uma ação. Esta ação é especificada no momento da definição da abstração para ser posteriormente executada. Abstrações são utilizadas para expressar a semântica de funções e procedimentos em linguagens de programação. As regras (5.36) a (5.41) definem a notação para este fim.

$$\frac{B, t, b, s \vdash y \triangleright abstraction(a, t_0, b_0) \quad t_0, b_0, s \vdash a \triangleright o', t', b', s'}{B, t, b, s \vdash enact y \triangleright o', t', b', s'} \quad (5.36)$$

$$B, t, b, s \vdash abstraction a \triangleright abstraction(a, \{\}, \{\}) \quad (5.37)$$

$$\frac{B, t, b, s \vdash y_1 \triangleright abstraction(a, \{\}, b_0) \quad B, t, b, s \vdash y_2 \triangleright d}{B, t, b, s \vdash y_1 \text{ with } y_2 \triangleright abstraction(a, \{0 \mapsto d\}, b_0)} \quad (5.38)$$

$$\frac{B, t, b, s \vdash y_1 \triangleright abstraction(a, t_0, b_0) \quad B, t, b, s \vdash y_2 \triangleright d \quad t_0 \neq \{\}}{B, t, b, s \vdash y_1 \text{ with } y_2 \triangleright abstraction(a, t_0, b_0)} \quad (5.39)$$

$$\frac{B, t, b, s \vdash y \triangleright abstraction(a, t_0, \{\})}{B, t, b, s \vdash closure y \triangleright abstraction(a, t_0, b)} \quad (5.40)$$

$$\frac{B, t, b, s \vdash y \triangleright abstraction(a, t_0, b_0) \quad b_0 \neq \{\}}{B, t, b, s \vdash closure y \triangleright abstraction(a, t_0, b_0)} \quad (5.41)$$

Em (5.36) definimos a chamada a abstrações. Em “enact y ” a operação referenciada por y especifica a abstração a ser referenciada. Os dados transitórios e bindings são repassados à ação incorporada, que produz a tripla (t', b', s') além de um resultado definido por o' . A criação de uma abstração é definida em 5.37. A abstração definida não possui dados transitórios nem bindings. As regras (5.38) e (5.39) estabelecem a semântica da operação ‘ y_1 with y_2 ’, permitindo a passagem de um dado simples resultante da operação y_2 , para a abstração representada por y_1 . Bindings

podem ser fornecidos às abstrações, através da operação ‘closure y ’. Este comportamento é determinado pelas regras (5.40) e (5.41).

Ressaltamos que, pelas regras apresentadas, a ligação entre nomes de métodos e ações no formalismo (representado por B) segue a cadeia dinâmica. É possível definir uma versão de Semântica de Ações Orientada a Objetos que possua bindings de métodos estaticamente definidos. Para isto basta incluir o componente B na definição das *closures*. A escolha por bindings de métodos (*Meta-Bindings*) dinâmicos responde à natureza dinâmica da configuração das classes dentro de uma especificação.

Exemplo 5.3.10 (Uso da notação reflexiva em uma Declaração de função)

```

Class FunctionDeclaration
  extending Declaration
  using E:Expression, T:Type, FP:FormalPar
  syntax:
    Dec ::= "func" I "(" FP ")" ":" E
  semantics:
    elaborate[["func" I "(" FP ")" ":" E]]=
      recursively bind I
        to closure abstraction of
          furthermore elaborateFP FP
        hence
          evaluate E then give the value
End Class

```

No exemplo 5.3.10, a declaração de função cria um binding entre o identificador definido por I e uma abstração de função. A abstração têm o efeito de produzir o resultado da expressão E , utilizando as informações de atuais de dados transitórios e bindings.

Exemplo 5.3.11 (Uso da notação reflexiva em uma chamada de função)

```

Class FunctionExpression
  extending Expression
  using I:Identifier, AP:ActualPars
  syntax:
    Com ::= I "(" AP ")"
  semantics:
    execute[[I "(" AP ")"]]=
      evaluateAP AP
    then
      enact(the function bound to I
        with the value)
End Class

```

No exemplo 5.3.11, definimos a chamada a funções. A expressão determinada por AP é

avaliada sendo o valor final repassado à abstração. A operação *enact* é responsável por produzir o resultado final de dados transitórios, bindings e memória com base na execução da abstração.

5.3.8 Operações Híbridas

As operações híbridas interagem com mais de um tipo de atributo dos (t, b, s) . As regras utilizadas para este fim são descritas de (5.42) a (5.45).

$$\frac{B, \{\}, b, s \vdash a_1 \triangleright o_1, t_1, b_1, s_1}{(0 \mapsto true) \cdot t, b, s \vdash a_1 \text{ else } a_2 \triangleright o_1, t_1, b_1, s_1} \quad (5.42)$$

$$\frac{B, \{\}, b, s \vdash a_2 \triangleright o_2, t_2, b_2, s_2}{B, (0 \mapsto false) \cdot t, b, s \vdash a_1 \text{ else } a_2 \triangleright o_2, t_2, b_2, s_2} \quad (5.43)$$

$$\frac{B, t, \{k \mapsto d\} \odot b, s \vdash y \triangleright d \quad d : \text{abstraction}}{B, t, b, s \vdash \text{recursively bind } k \text{ to } y \triangleright \text{completed}, \{\}, \{k \mapsto d\}, s} \quad (5.44)$$

$$\frac{c : \tau \leq \text{cell} \quad c \notin \text{dom } s}{B, t, b, s \vdash \text{allocate a } r \triangleright \text{completed}, \{0 \mapsto c\}, \{\}, \{c \mapsto \text{uninitialized}\} \oplus s} \quad (5.45)$$

A operação definida por “ $A_1 \text{ else } A_2$ ”, determina a execução condicionada da primeira ou segunda sub-operação. A operação principal recebe como entrada a tripla $((0 \mapsto true) \cdot t, b, s)$ ou $((0 \mapsto false) \cdot t, b, s)$. A notação $(0 \mapsto d) \cdot t$ define que o primeiro dado transitório do conjunto t deve ter valor *true* ou *false*. No primeiro caso a sub-ação A_1 é executada produzindo (t_1, b_1, s_1) , como descrito em 5.42. Caso contrário A_2 é executada, resultando em (t_2, b_2, s_2) conforme (5.43).

Em (5.44) definimos o comportamento de “recursively bind k to y ”. Esta operação possibilita a criação de bindings recursivos. A operação produtora y deve prover um dado d previamente criado no ambiente corrente disponível na tripla de entrada (t, b, s) . O resultado é $(\{\}, \{k \mapsto d\}, s)$, indicando que não serão gerados novos dados transitórios e a memória permanece inalterada. Todavia, um único binding é criado, estabelecendo assim um mapeamento entre o identificador k e o dado d (abstração).

A operação “allocate a r ”, conforme a regra 5.45, possibilita a alocação uma célula de memória. Dada a tripla (t, b, s) , esta operação resulta em $(\{0 \mapsto c\}, \{\}, \{c \mapsto \text{uninitialized}\} \oplus s)$, indicando respectivamente que: uma célula c está disponibilizada como dado transitório às demais ações; não há bindings sendo criados; esta nova célula de memória está disponível para ser utilizada (status *uninitialized*).

Exemplo 5.3.12 (Uso da notação híbrida em Declaração de variáveis)

Class Type

```
syntax:
  T ::= "boolean" | "integer"
semantics:
  allocate-for-type _ : T -> Action
  allocate-for-type[["boolean"]] =
    allocate a cell [ truth-value ]
  allocate-for-type[["integer"]] =
    allocate a cell [ integer ]
```

End Class

Class VariableDeclaration

```
extending Declaration
using E:Expression, T:Type
syntax:
  Dec ::= "var" I ":" T
semantics:
  elaborate[["var" I ":" T]] =
    allocate-for-type
  then
    bind token I to the given cell
```

End

As classes Type e VariableDeclaration estão envolvidas na declaração de variáveis. A operação 'allocate-for-type _' disponibilizada na classe Type, utiliza a operação híbrida allocate. Na especificação da declaração de variáveis, a chamada de allocate-for-type T, além de alocar uma célula de memória, produz a mesma como dado transitório para ser utilizada. Esta é então rotulada com #1, sendo repassada para a última sub-operação que efetua o binding.

5.3.9 Sub-classes de State

As classes definidas pelo usuário em Semântica de Ações Orientada a Objetos são, como já vimos, sub-classes de State. Por consequência, elas *herdam* os atributos e operações descritos nas seções anteriores.

Para cada método (equação semântica) na forma $f \llbracket H \rrbracket = a$, sendo H uma *syntax-tree* [Mos92] e a uma *operação* ou ação, especificada nas seções 5.3.4, 5.3.5, 5.3.6, definimos as seguintes regras:

$$\frac{(f \llbracket H \rrbracket) \in \text{dom}(B), \quad B, t, b, s \vdash A[H/h] \triangleright o, t', b', s'}{B, t, b, s \vdash f \llbracket h \rrbracket \triangleright o, t', b', s'} \quad (5.46)$$

$$\frac{}{B \vdash f[[H]] = A \triangleright B[f[[H]] \rightarrow A]} \quad (5.47)$$

A regra 5.46 define que uma equação semântica no formato $f[[h]]$ produz o resultado o, t', b', s' se existir uma definição desta equação no ambiente de definição de métodos B , de maneira que, substituindo a *syntax-tree* H por h , a ação A definida sobre a equação f pode ser executada. Em 5.47 determinamos que cada definição de equação no formato ' $f[[H]] = A$ ' deve ser inserida no ambiente B . A notação $B[f[[H]] \rightarrow A]$ indica que uma nova definição foi adicionada no ambiente de métodos (Meta-Bindings) B .

CAPÍTULO 6

ESTUDO DE CASO: A LINGUAGEM PURPLE

Neste capítulo apresentamos a Semântica Orientada a Objetos de uma linguagem de programação. PURPLE é uma linguagem de programação imperativa baseada em IMP [Win93] [Wat91] e SPECIMEN [Mou93]. Criada neste trabalho, ela visa demonstrar a utilização da Semântica de Ações Orientada a Objetos. PURPLE contém diversas características comuns a este tipo de linguagem incluindo expressões (aritméticas, lógicas, etc.), comandos (atribuições, condições, repetições, etc.) e declarações.

6.1 Sintaxe da Linguagem PURPLE

Definimos a seguir a sintaxe abstrata da linguagem PURPLE. O formato básico de um programa nesta linguagem é definido a partir do símbolo não-terminal `Program`:

- (1) `Program =`
 `[[ "declare" Declaration "used-in" Command ]]`.
- (2) `Declaration =`
 `[[ "var" Identifier ":" Type <"=" Expression>? ] ] |`
 `[[ "const" Identifier ":" Type "=" Expression ] ] .`
- (3) `Type =`
 `"boolean" | "integer"`

Em (1) definimos que um programa é o texto `declare` seguido de uma declaração. Esta vale dentro do bloco de comandos definidos sob `used-in`. Definimos então variáveis e constantes, pela regra (2), indicando seu tipo (`Type`), conforme (3). Uma expressão define um valor inicial para constantes e variáveis, sendo que para variáveis esta definição inicial é opcional.

Exemplo 6.1.1 (Declarações na linguagem PURPLE)

Vamos exemplificar as seguintes declarações:

“var a : boolean = true” cria uma variável booleana a, com valor inicial “true”;

“var b : integer = c * 2” cria uma variável inteira b, que tem como valor inicial o resultado da expressão c * 2;

“const d : boolean = false” cria uma constante booleana identificada por d, com valor inicial “false”;

“const e : integer = 5” cria uma constante inteira identificada por e, com valor inicial “5”.

Exemplo 6.1.2 (Estrutura geral de um programa na linguagem PURPLE)

```
declare
  var x : integer
used-in
  declare
    const y : integer = 3
  used-in
    x := y * y
```

O exemplo 6.1.2 mostra um programa simples na linguagem PURPLE, que atribui a uma variável x o quadrado do valor de uma constante representada por y. Ao ser declarada, a variável x torna-se válida em toda extensão do bloco used-in. No escopo da variável x, o comando definido é uma nova estrutura “declare ... used-in ...”. Uma constante y é criada dentro do novo bloco, com valor igual a 3.

A criação de um comando representado pelo bloco “declare ... used-in ...”, e da atribuição “x := y * y” são possíveis em razão das definições de Command, como apresentado a seguir.

- (1) Command =
- [[“declare” Declaration “used-in” Command]] |
 - [[Identifier “:=” Expression]] |
 - [[“if” Expression “then” Command { “else” Command }? “end-if”]] |
 - [[“repeat” Command “until” Expression]] |
 - [[Command “;” Command]].

Os comandos da linguagem, conforme as definições de Command em (1) são respectivamente:

- *Declaração seguida de comandos*: Um novo escopo é definido com o uso do token “declare” seguido da declaração (Declaration). O bloco de comandos representado por Command referente a este escopo é definido sob “used-in”;

- *Atribuição*: Assinala o resultado de uma expressão representada por Expression a uma variável definida por Identifier;
- *Condicional*: Corresponde ao comando "if...end-if" que testa o valor de uma expressão representada por Expression e executa o primeiro comando, caso o resultado da expressão seja "true", ou o segundo comando caso contrário. Ambos os comandos são representados pelo símbolo não-terminal Command. O bloco "else" é opcional;
- *Repetição*: O comando "repeat...until..." representa a repetição nesta linguagem. Um comando representado por Command é executado até a expressão Expression seja verdadeira;
- *Seqüência*: Os comandos representados por Command são separados por ";".

Exemplo 6.1.3 (Comandos na linguagem PURPLE)

Vamos exemplificar os seguintes comandos:

"x:= y * 2" corresponde a uma atribuição do valor resultante da expressão y * 2, na variável x;

"if x=2 then b:=true else b:=false end-if" é um comando de seleção. Se o valor da variável x é 2, o valor "true" é atribuído a b, caso contrário o valor "false" é atribuído a b;

"repeat x:=x+1 until x=10" o resultado de x+1 é atribuído variável x, até que o valor da mesma chegue a 10;

"x:=1; y:=2" os comandos são executados em seqüência, separados por ";".

As expressões presentes nos comandos, representadas pelo não-terminal Expression, estão definidas conforme as regras seguintes.

- (1) Expression =
- [["true"]] |
 - [["false"]] |
 - [[Numeral]] |
 - [[Identifier]] |
 - [[Expression "+" Expression]] |
 - [[Expression "-" Expression]] |
 - [[Expression "*" Expression]] |
 - [[Expression "<" Expression]] |
 - [[Expression "=" Expression]] .

- (2) Identifier =
 $\llbracket \text{Letter} \langle \text{Letter} \mid \text{Digit} \rangle^* \rrbracket$.
- (3) Numeral =
 $\llbracket \text{Digit} \langle \text{Digit} \rangle^* \rrbracket$.

As expressões na linguagem PURPLE estão definidas pelas regras em (1), anteriores. Os tokens “true” e “false” correspondem respectivamente aos valores true e false em expressões lógicas. Uma expressão pode ser um número inteiro (Numeral) ou um identificador (Identifier). As cinco últimas regras definidas em (1) definem as expressões aritméticas utilizando os operadores “+”, “-” e “*”, e as expressões lógicas utilizando os operadores “<” e “=”.

As regras (2) e (3) definem o formato de identificadores e números pela especificação de Identifier e Numeral, respectivamente.

6.2 Um programa para cálculo do fatorial na Linguagem PURPLE

O programa apresentado a seguir é baseado na linguagem PURPLE e tem o objetivo de calcular o fatorial de um número x declarado inicialmente com o valor 5.

```

declare
    var x : integer = 5
used-in
    declare
        var fat : integer = 1
    used-in
        repeat
            fat := fat * x;
            x := x - 1
        until x=1

```

Uma variável x , inteira, é declarada inicialmente com valor 5. No bloco “declare ... used-in ...” mais interno uma variável fat é criada com valor 1. Uma repetição é efetuada pelo comando “repeat...until...”, sendo o valor do fatorial de x armazenado em fat .

6.3 Semântica de Ações Orientada a Objetos da Linguagem PURPLE

Com base na sintaxe abstrata apresentada nas Seções anteriores, vamos construir uma *hierarquia* de classes para representar a semântica da linguagem PURPLE. As regras de especificação das frases da linguagem (declarações, comandos, expressões) serão mapeadas para objetos que representem seus significados.

Inicialmente, precisamos definir os *sorts* que serão referenciados pelas classes, ao longo da especificação.

6.3.1 Entidades Semânticas

Os *sorts* de (1) a (5) definidos a seguir serão utilizados nas classes. Em (1) definimos o *sort value* indicando que os valores manipulados nas classes serão inteiros ou valores-verdade. O *sort token*, apresentado em (2) tem a mesma definição. Apresentamos em (3) os elementos do *sort bindable*, sendo valores (no caso de constantes) ou células (no caso de variáveis). Em (4) definimos o *sort cell* para armazenar inteiros ou valores-verdade. Os elementos que podem ser armazenados nas células, representados pelo *sort storable*, conforme (5), são números inteiros e valores-verdade.

- (1) `value = integer | truth-value.`
- (2) `token = value.`
- (3) `bindable = value | cell.`
- (4) `cell = cell [ truth-value ] | cell [ integer ].`
- (5) `storable = value.`

6.3.2 Classes

Nesta Seção vamos apresentar a especificação da linguagem PURPLE baseada em uma hierarquia de Classes. Utilizaremos alguns diagramas para expressar hierarquia, adotando uma notação na forma de árvore.

Na definição de PURPLE, adotamos uma abordagem *bottom-up* onde construiremos a especificação da linguagem com base na composição das classes definidas para as frases da mesma.

Para definir a especificação completa, inicialmente vamos apresentar algumas classes auxiliares que não aparecem na hierarquia da definição da linguagem PURPLE, porém são instanciadas pelas classes principais. Estas classes serão apresentadas a seguir.

6.3.2.1 Identificadores, Numerais e Tipos

As classes que representam identificadores, numerais e tipos (boolean e integer) na linguagem PURPLE, têm a seguinte definição:

```
Class Identifier
  syntax:
    Id ::= letter [ letter | digit ]*
End Class
```

A classe `Identifier` representa os nomes de constantes e variáveis na linguagem. Esta classe possui apenas representação sintática (utiliza apenas a seção `syntax`), estabelecendo a estrutura dos nomes como podemos verificar na definição de `Id`.

```
Class Numeral
  syntax:
    N ::= digit+
  semantics:
    valuation _ : N -> integer
End Class
```

A classe `Numeral` tem representação sintática (definida em `syntax`), conforme a definição do símbolo não-terminal `N`, definindo a estrutura dos números inteiros. Na seção semântica (`semantics`), apresentamos um método (função semântica) denominado `valuation _`, que especifica o significado de um numeral. Este método segue a definição de [Wat91].

```
Class Type
  syntax:
    T ::= "boolean" | "integer"
  semantics:
    allocate-for-type _ : T -> Action
    allocate-for-type[[ "boolean" ]] =
      allocate a cell [ truth-value ]
    allocate-for-type[[ "integer" ]] =
      allocate a cell [ integer ]
End Class
```

A seção sintática de `Type`, define que os tipos da linguagem como `"boolean"` e `"integer"`. Esta classe é utilizada nas declarações de variáveis, onde é necessário alocar um espaço em memória. A semântica dos tokens que definem os tipos, como visto em (`semantics`) é reservar uma célula de memória, dependendo do tipo definido. O método `'allocate-for-type _'` estabelece duas ações possíveis, correspondentes aos dois tipos de dados da linguagem: reservar um valor-verdade ou um valor inteiro.

A Seção seguinte apresenta as classes que representam as declarações na linguagem `PURPLE`.

6.3.2.2 Declarações

A linguagem `PURPLE` permite a criação de constantes e variáveis. Podemos verificar a estrutura das classes que denotam estas possíveis declarações na Figura 6.1.

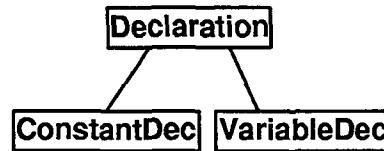


Figura 6.1: Hierarquia de classes de Declarações na Linguagem PURPLE

```

Class Declaration
    using I : Identifier, E:Expression
    syntax:
        Dec
    semantics:
        elaborate _ : Dec -> Action
End Class

```

A Classe Declaration instancia objetos das classes Identifier e Expression (definida nas próximas Seções). Estes objetos não são utilizados na classe Declaration, mas são definidos aqui para serem disponibilizados para suas sub-classes. Introduzimos na parte sintática o símbolo não-terminal Dec a ser redefinido pelas sub-classes. Na seção semântica a função ‘elaborate _’ é especificada indicando que uma declaração está mapeada para uma ação.

```

Class ConstantDec
    extending Declaration
    syntax:
        Dec ::= "const" I ":" T "=" E
    semantics:
        elaborate[["const" I ":" T "=" E]] =
            evaluate E then
            bind I to the value
End Class

```

Uma declaração de constante (ConstantDec) utiliza os objetos I:Identifier e E:Expression da super-classe. Redefinimos Dec, especificando a estrutura de uma declaração de constante. A semântica da declaração de constante consiste em efetuar um *binding* do identificador I ao valor produzido pela chamada do método evaluate do objeto E. Este comportamento é estabelecido pelo método ‘elaborate’, redefinido para os fins de ConstantDec.

```

Class VariableDec
  extending Declaration
  using T:Type
  syntax:
    Dec ::= "var" I ":" T ["=" E]
  semantics:
    elaborate[["var" I ":" T]]=
      allocate-for-type T
    then
      bind token I to the cell #0
    elaborate[["var" I ":" T "=" E]]=
      evaluate E
    and
      allocate-for-type T
    then
      bind I to the cell #1
    and
      store the value #0 in the cell #1
End Class

```

A classe `VariableDec` instancia um objeto `T:Type`. A estrutura de uma declaração de variável é redefinida em `Dec`, na parte sintática da classe. Duas possibilidades de declaração de variáveis são disponibilizadas na seção de semântica pela sobrecarga do método ‘`elaborate`’. A primeira contempla a declaração de variáveis, sem considerar a especificação de valor inicial para a mesma. Neste caso, um *binding* é efetuado entre o identificador `I` e uma célula de memória, sendo que o método ‘`allocate-for-type`’ do objeto `T` reserva o espaço em memória para este fim. A segunda versão de ‘`elaborate`’ expressa da mesma forma a criação do binding; todavia, um valor é armazenado na célula de memória reservada, resultante da chamada do método ‘`evaluate`’ do objeto `E`, definido na classe base (`Declaration`).

Na Seção seguinte apresentaremos a especificação dos comandos da linguagem PURPLE.

6.3.2.3 Comandos

A hierarquia de comandos da linguagem PURPLE está definida conforme a Figura 6.2. A classe principal dos comandos da linguagem é `Command`.

```

Class Command
  syntax:
    Com
  semantics:
    execute _ : Com -> Action
End Class

```

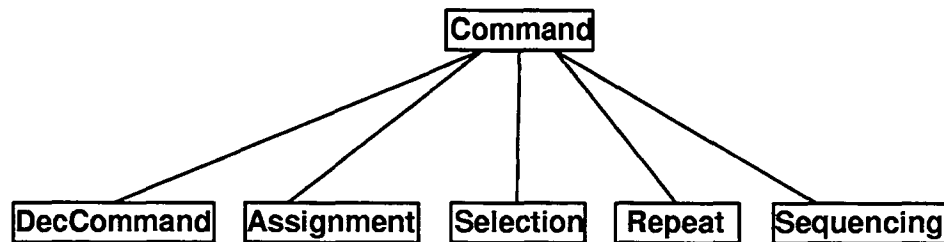


Figura 6.2: Hierarquia de classes de Comandos na Linguagem PURPLE

Estruturamos `Command` contendo, na parte sintática, a definição de um símbolo não-terminal `Com` a ser redefinido pelas sub-classes. Na parte semântica, o método `'execute _'` estabelece o mapeamento da estrutura sintática `Com` para uma ação correspondente.

```

Class DecCommand
  extending Command
  using D:Declaration, C:Command
  syntax:
    Com ::= "declare" D "used-in" C
  semantics:
    execute[["declare" D "used-in" C]] =
      furthermore elaborate D
      hence
      execute C
End Class
  
```

A classe `DecCommand` instancia um objeto `D:Declaration` e um objeto `C:Command`. O símbolo não-terminal `Com` é redefinido na seção de sintaxe, estabelecendo a estrutura de declarações seguidas de comandos. O método `'execute'` é redefinido na seção de semântica tendo a ação de efetuar a declaração `D` (utilizando seu método `'elaborate'`) e executar o comando `C` (utilizando seu método `'execute'`) no novo escopo.

```

Class Assignment
  extending Command
  using I:Identifier, E:Expression
  syntax:
    Com ::= I ":@" E
  semantics:
    execute[[I ":@" E]] =
      evaluate E
      then
      store the value in the cell bound to I
End Class
  
```

O comando de atribuição é representado pela classe `Assignment`. Os objetos `I:Identifier` e `E:Expression` são instanciados por esta classe. Na seção de sintaxe a estrutura da atribuição é definida e, na parte semântica da classe, uma nova versão de ‘execute’ é empregada para denotar a atribuição. A ação utilizada neste método efetua o armazenamento de um valor na célula de memória correspondente a `I`.

Class Selection

```

    extending Command
    using E:Expression, C1:Command, C2:Command
    syntax:
        Com ::= "if" E "then" C1 [ "else" C2 ] "end-if"
    semantics:
        execute[[ "if" E "then" C1 "end-if" ]]=
            evaluate E
            then
                execute C1
            else
                complete
        execute[[ "if" E "then" C1 "else" C2 "end-if" ]]=
            evaluate E
            then
                execute C1
            else
                execute C2

```

End Class

Na definição da classe `Selection` são instanciados três objetos: `E:Expression`, `C1:Command` e `C2:Command`. Na parte sintática definimos o formato da estrutura de seleção, informando a possibilidade da omissão do segundo comando (["else" C2]).

Por esta razão, na parte semântica definimos duas novas possibilidades para o método ‘execute’. Em ambas a expressão representada pelo objeto `E` produz um valor-verdade mediante a chamada do método ‘evaluate’. Na primeira equação, existe apenas a possibilidade de execução do primeiro comando (`C1`). Na segunda, em caso positivo o primeiro é executado (`C1`), e caso contrário, o segundo é executado (`C2`).

```

Class Repeat
    extending Command
    using E:Expression, C:Command
    syntax:
        Com ::= "repeat" C "until" E
    semantics:
        execute[[ "repeat" C "until" E ]]=
            unfolding
                execute C
            and then
                evaluate E
            then
                complete
            else
                unfold
End Class

```

A estrutura de repetição da linguagem PURPLE é representada pela classe *Repeat*. Seguindo a estrutura das demais sub-classes de *Command*, redefinimos a sintaxe pela adição da estrutura do comando de repetição. Dois objetos são instanciados (*E:Expression*, *C:Command*) os quais produzem resultados pela chamada de seus métodos ‘*evaluate*’ e ‘*execute*’, respectivamente. A semântica da estrutura de repetição consiste na reexecução do comando *C* até que a expressão *E* produza valor verdadeiro.

```

Class Sequencing
    extending Command
    using C1:Command, C2:Command
    syntax:
        Com ::= C1 ";" C2
    semantics:
        execute[[ C1 ";" C2 ]]=
            execute C1 and then execute C2
End Class

```

A seqüenciação de comandos é expressa pela classe *Sequencing*. Os comandos são separados por “;”, sendo esta regra definida na parte sintática. Uma nova versão ‘*execute*’ é definida, indicando a execução seqüenciada dos objetos *C1* e *C2*, previamente instanciados nesta classe.

Note que na definição de *Assignment*, *Selection* e *Repeat*, foram instanciados objetos da classe *Expression*. A definição das expressões da linguagem PURPLE pode ser vista a seguir.

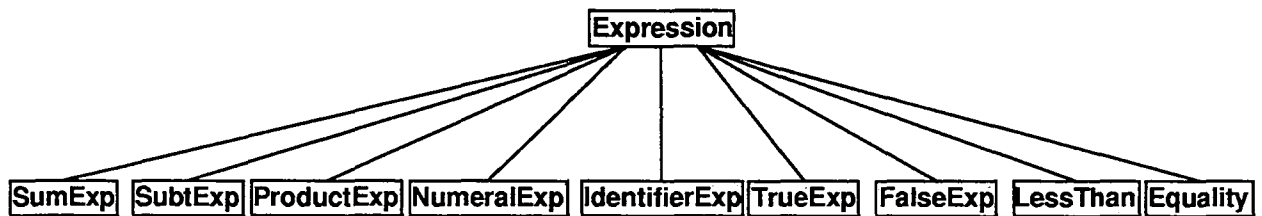


Figura 6.3: Hierarquia de classes de Expressões na Linguagem PURPLE

6.3.2.4 Expressões

Expressões aritméticas e lógicas compõem a linguagem. A hierarquia de expressões está definida conforme a Figura 6.3.

```

Class Expression
  syntax:
    Exp
  semantics:
    evaluate _ : Exp -> Action
End Class
  
```

A classe principal da hierarquia de expressões é `Expression`. Seguindo o mesmo padrão das classes `Declaration` e `Command` das Seções anteriores, definimos o símbolo não-terminal `Exp` que serve de base para a especificação da sintaxe das expressões. Na seção semântica definimos um método `'evaluate _'`, a ser especializado em cada sub-classe.

As três classes a seguir definem a soma, subtração e produto entre expressões.

```

Class SumExp
  extending Expression
  using E1:Expression, E2:Expression
  syntax:
    Exp ::= E1 "+" E2
  semantics:
    evaluate[[ E1 "+" E2 ]] =
      evaluate E1
      and
      evaluate E2
    then
      give sum(the integer#0, the integer#1)
End Class
  
```

```

Class SubtExp
    extending Expression
    using E1:Expression, E2:Expression
    syntax:
        Exp ::= E1 "-" E2
    semantics:
        evaluate[[ E1 "-" E2 ]] =
            evaluate E1
            and
            evaluate E2
        then
            give subtraction (the integer#0, the integer#1)
End Class

```

```

Class ProductExp
    extending Expression
    using E1:Expression, E2:Expression
    syntax:
        Exp ::= E1 "*" E2
    semantics:
        evaluate[[ E1 "*" E2 ]] =
            evaluate E1
            and
            evaluate E2
        then
            give product(the integer#0, the integer#1)
End Class

```

Podemos verificar uma estrutura comum na definição das classes SumExp, SubtExp, ProductExp. A diferença sintática está na definição do operador binário utilizado - "+", "-" e "*" para a representação da soma, subtração e multiplicação respectivamente, entre expressões. Na parte semântica, em todos os casos os objetos E1 e E2 são instanciados e produzem valores mediante a execução do método 'evaluate _'. O resultado final é definido porém pelas funções auxiliares 'sum(_,_) ', 'subtraction(_,_) ' ou 'product(_,_) ' que efetuam as operações da soma, subtração ou multiplicação baseado nos valores das expressões.

A descrição das expressões acima segue a hierarquia imposta pela sintaxe da linguagem apresentada na página 66. Seria possível definir uma hierarquia diferente, do estilo apresentado em [MM01].

```

Class NumeralExp
    extending Expression
    using N:Numeral
    syntax:
        Exp ::= N
    semantics:
        evaluate [[ N ]] =
            give valuation N
End Class

```

A classe NumeralExp representa a semântica de um numeral em expressões. Um objeto N:Numeral é instanciado, e seu método valuation produz um número inteiro que representa o numeral.

```

Class IdentifierExp
    extending Expression
    using I:Identifier
    syntax:
        Exp ::= I
    semantics:
        evaluate [[ I ]] =
            give the value bound to I
        or
            give the value stored in the cell bound to I
End Class

```

Um identificador na linguagem PURPLE pode ser uma variável ou constante. A semântica do identificador é expressa pela classe IdentifierExp. Na parte sintática o token Exp é redefinido com a estrutura sintática de um identificador (I). O método 'evaluate' expressa a semântica da avaliação do identificador que produz o valor relacionado a este.

As classes seguintes definem a semântica das expressões lógicas.

```

Class TrueExp
    extending Expression
    syntax:
        Exp ::= "true"
    semantics:
        evaluate[[ "true" ]] =
            give true
End Class

```

```

Class FalseExp
    extending Expression
    syntax:
        Exp ::= "false"
    semantics:
        evaluate[[ "false" ]] =
            give false
End Class

```

As classes TrueExp e FalseExp definem a semântica das expressões representadas pelos *tokens* "true" e "false". Em ambos os casos o método 'evaluate' é redefinido indicando a ação que representa a semântica das frases.

```

Class LessThan
    extending Expression
    using E1:Expression, E2:Expression
    syntax:
        Exp ::= E1 "<" E2
    semantics:
        evaluate[[ E1 "<" E2 ]] =
            evaluate E1
            and
            evaluate E2
        then
            give less(the integer#0, the integer#1)
End Class

```

```

Class Equality
    extending Expression
    using E1:Expression, E2:Expression
    syntax:
        Exp ::= E1 "=" E2
    semantics:
        evaluate[[ E1 "=" E2 ]] =
            evaluate E1
            and
            evaluate E2
        then
            give equal(the integer#0, the integer#1)
End Class

```

As classes LessThan e Equality definem as expressões lógicas que utilizam os operadores "<" e "=", respectivamente. A estrutura destas classes é similar a que foi definida para as classes de expressões que utilizam operadores aritméticos (SumExp por exemplo). A função auxiliar

'less(,)' da classe LessExp produz um valor verdadeiro se o valor resultante da avaliação da expressão E1 for menor do que o resultado de E2 e falso em caso contrário. De maneira similar a função auxiliar 'equal(,)' é aplicada na classe EqualityExp.

6.3.2.5 A Linguagem

Definidas as hierarquias de declarações (Declaration), comandos (Command) e expressões (Expression) como vimos nas Seções anteriores, podemos apresentar a classe que representa a linguagem PURPLE.

```
Class PurpleLanguage
  using D:Declaration, C:Command
  syntax:
    Prog ::= "declare" D "used-in" C
  semantics:
    run _ : Prog -> Action
    run[[ "declare" D "used-in" C ]]
      elaborate D
    hence
      execute C
End Class
```

Dois objetos são instanciados nesta classe: D:Declaration e C:Command. O token Prog é definido na parte sintática da classe e a estrutura geral de um programa na linguagem PURPLE é estabelecida. Na seção de semântica, definimos um método 'run _', que mapeia um programa nesta linguagem para sua denotação (uma ação).

Com isto completamos a definição de PURPLE: uma linguagem imperativa simples. Nas próximas Seções introduziremos novas características à linguagem de forma a exemplificar o uso da extensão da especificação apresentada até o momento.

6.4 Estendendo a especificação - A Linguagem *X-Purple*

Apresentamos nesta Seção uma extensão da linguagem PURPLE denominada *X-Purple*. Utilizando partes da linguagem original podemos compô-la, criando a nova linguagem. Esta extensão visa incorporar duas estruturas fundamentais em linguagens de programação imperativa: Procedimentos e Funções. Inicialmente mudanças na sintaxe da linguagem são necessárias.

6.4.1 Sintaxe da Linguagem *X-Purple*

A criação de procedimentos e funções requer alguns incrementos na estrutura original de PURPLE. As novas definições estão apresentadas a seguir.

- (1) Declaration =
 ... |
 [["proc" Identifier "(" Formal-Par ")" "=" Command]] |
 [["func" Identifier "(" Formal-Par ")" "=" Expression]].
- (2) Formal-Par =
 [["var" Identifier ":" Type]].

Em (1) adicionamos mais duas regras sintáticas contemplando a declaração de procedimentos e funções. Utilizando a mesma estrutura definida em [Wat91] estabelecemos que um procedimento executa um comando (Command) e uma função retorna um valor resultante da avaliação de uma expressão (Expression). As reticências (...) são utilizadas para indicar que a linguagem está sendo estendida.

Em *X-Purple*, procedimentos e funções admitem um único parâmetro definido conforme (2). A definição do parâmetro obedece a mesma estrutura definida para a declaração de variáveis.

- (1) Command =
 ... |
 [["call" Identifier "(" Actual-Par ")"]].
- (2) Expression =
 ... |
 [[Identifier "(" Actual-Par ")"]].
- (3) Actual-Par =
 [[Expression]].

Além da declaração, devemos acrescentar na sintaxe as regras para a chamada de procedimentos e funções. Em (1) definimos a chamada a procedimentos, adicionando esta regra às definições de Command. A chamada a funções é especificada modificando as definições de Expression, conforme (2). Os parâmetros reais são definidos por mais uma regra conforme (3).

6.4.2 Declaração de Funções e Procedimentos

A semântica da declaração de procedimentos e funções é expressa por meio da extensão da classe Declaration definida para a linguagem PURPLE. Modificamos a hierarquia de declarações para adicionar duas novas classes: ProcedureDec e FunctionDec, conforme a Figura 6.4.

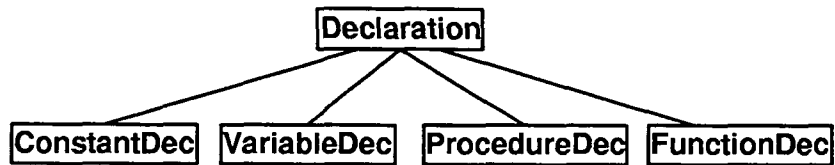


Figura 6.4: Hierarquia de classes de Declarações na Linguagem *X-Purple*

```

Class ProcedureDec
  extending Declaration
  using C:Command, FP:FormalPar
  syntax:
    Dec ::= "proc" I "(" FP ")" ":" C
  semantics:
    elaborate[["proc" I "(" FP ")" ":" C]]=
      recursively bind I
        to closure abstraction of
          furthermore elaborateFP FP
        hence
          execute C
End Class

```

A classe de declaração de procedimentos (ProcedureDec) instancia dois objetos: *C:Command* e *FP:FormalPar*. Uma nova sintaxe é definida para as declarações contemplando os procedimentos. Na seção de semântica um objeto *I:Identifier* (definido na classe *Declaration*) produz um identificador a ser ligado à abstração de procedimento.

```

Class FunctionDec
  extending Declaration
  using T:Type, FP:FormalPar
  syntax:
    Dec ::= "func" I "(" FP ")" ":" E
  semantics:
    elaborate[["func" I "(" FP ")" ":" E]]=
      recursively bind I
        to closure abstraction of
          furthermore elaborateFP FP
        hence
          evaluate E then give the value
End Class

```

A declaração de funções, definida pela classe *FunctionDec* segue a mesma estrutura da classe anterior. Todavia, uma expressão *E* (definida em *Declaration*) é utilizada na criação da abstração.

As duas classes anteriores utilizam um objeto `FormalPar` que define os parâmetros para os procedimentos e funções. A definição desta classe está especificada a seguir.

6.4.3 Parâmetros Formais e Parâmetros Reais

Os semântica dos parâmetros de procedimentos e funções são expressas pelas classes `FormalPar` e `ActualPar`, que definimos da seguinte forma:

```
Class FormalPar
  using I:Identifier, T:Type
  syntax:
    FP ::= "var" I ":" T
  semantics:
    elaborateFP _ : FP -> Action
    elaborateFP [[ "var" I ":" T ]] =
      allocate-for-type T
    then
      bind I to the cell #0
End Class
```

A semântica dos parâmetros no momento da declaração dos procedimentos e funções é expressa pela classe `FormalPar`. A sintaxe para declaração do parâmetro está definida pela regra relacionada ao token `FP`, na parte sintática da classe. Na seção de semântica definimos um método `elaborateFP` para expressar a declaração do parâmetro. A semântica efetivamente é efetuar um binding entre o identificador representado por `I` e a célula de memória alocada.

```
Class ActualPar
  using E:Expression
  syntax:
    AP ::= E
  semantics:
    evaluateAP _ : AP -> value
    evaluateAP [[ E ]] =
      evaluate E
End Class
```

A classe `ActualPar` define a semântica dos parâmetros no momento da chamada de procedimentos ou aplicação de funções. O parâmetro real é tão somente uma expressão, como definido na regra do símbolo não-terminal `AP` na seção de sintaxe. A semântica do parâmetro real é, por sua vez, a semântica da expressão `E`, provida pelo método `evaluate`.

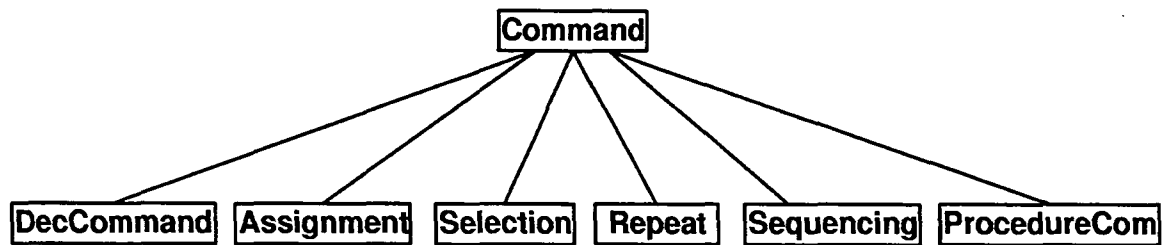


Figura 6.5: Hierarquia de classes de Comandos na Linguagem *X-Purple*

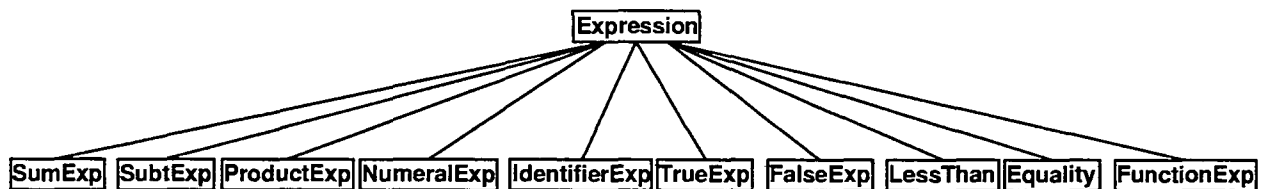


Figura 6.6: Hierarquia de classes de Expressões na Linguagem *X-Purple*

6.4.3.1 Chamadas a procedimentos e Aplicação de funções

A classe `ProcedureCom` é definida nesta linguagem como uma extensão da classe `Command`. A nova hierarquia de comandos pode ser vista na Figura 6.5. Da mesma maneira `FunctionExp` modifica a hierarquia de expressões (`Expression`), conforme a Figura 6.6.

```

Class ProcedureCom
  extending Command
  using I:Identifier, AP:ActualPar
  syntax:
    Com ::= "call" I "(" AP ")"
  semantics:
    execute[["call" I "(" AP ")"]] =
      evaluateAP AP
    then
      enact ( the procedure bound to I with the value )
End Class
  
```

A classe que representa a chamada de procedimentos (`ProcedureCom`) instancia dois objetos: `I:Identifier` e `AP:ActualPar`. A semântica de uma chamada a procedimentos é especificada na redefinição do método ‘`execute`’, definido inicialmente em `Command`. O comando definido pela abstração referenciada pelo identificador `I` é executado, dado um valor de entrada resultante da execução do método `evaluateAP` do objeto `AP`.

```

Class FunctionExp
  extending Expression
  using I:Identifier, AP:ActualPars
  syntax:
    Com ::= I "(" AP ")"
  semantics:
    evaluate[[I "(" AP ")"]]=
      evaluateAP AP
    then
      enact(the function bound to I with the value)
End Class

```

A classe FunctionExp que representa a aplicação de função tem estrutura similar a da classe ProcedureCom. Com esta definição, podemos então representar a nova linguagem.

6.4.4 A Linguagem Estendida

Para a criação da especificação da nova linguagem utilizamos algumas facilidades proporcionadas pela hierarquia de classes. O Objeto Declaration foi estendido para representar procedimentos e funções com base nas classes ProcedureDec e FunctionDec. A chamada a procedimentos foi incorporada a Command pela criação da classe ProcedureCom e a aplicação de funções faz parte do objeto Expression, pela definição da classe FunctionExp. Apenas duas classes foram criadas sem uso da especialização: FormalPar e ActualPar.

Como a definição das novas classes foi efetuada na maioria das vezes pela especialização de classes já existentes, podemos especificar a nova linguagem da seguinte forma:

```

Class X-PurpleLanguage
  using D:Declaration, C:Command
  syntax:
    Prog ::= "declare" D "used-in" C
  semantics:
    run _ : Prog -> Action
    run[[ "declare" D "used-in" C ]]
      elaborate D
    hence
      execute C
End Class

```

Portanto, definimos o protótipo final da classe X-PurpleLanguage de maneira idêntica ao protótipo da linguagem original.

CAPÍTULO 7

CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho apresentamos uma nova abordagem para Semântica de Ações, com o objetivo principal de prover mecanismos para o reuso de especificações. Utilizando as idéias propostas nos trabalhos de [DM01] e [MM01], que estabelecem métodos para a divisão da especificação em partes, introduzimos a utilização da orientação a objetos ao mesmo contexto. Uma notação foi estabelecida, respeitando a estrutura das descrições em Semântica de Ações [Mos92], e sua semântica também foi definida.

Semântica de Ações Orientada a Objetos constitui uma nova forma de organizar especificações escritas em Semântica de Ações. O estudo de caso apresentado no Capítulo 6 mostrou a aplicação desta abordagem a uma linguagem imperativa simples. Neste estudo de caso definimos uma hierarquia de classes para representar a semântica da linguagem Purple e utilizando conceitos da orientação a objetos, estabelecemos o reuso e a extensão da especificação criando uma nova linguagem, denominada *X-Purple*. Acreditamos ter tido sucesso nesta tarefa.

Esperamos que, com a crescente utilização das técnicas da orientação a objetos, este formalismo ou outros semelhantes, baseados na mesma idéia, possam ser empregados na representação formal de linguagens de programação.

Podemos sumarizar nossa contribuição a partir dos seguintes itens:

- A notação para escrita de especificações em nosso paradigma é de simples entendimento. A estrutura sintática para definição das classes está padronizada de acordo com algumas linguagens de programação orientadas a objetos como C++ e Java.
- O reuso de especificações é efetuado pela instanciação de objetos que representam as partes da linguagem. A própria definição de objetos sugere a divisão da descrição da linguagem em partes. Cada objeto é definido e uma interface é estabelecida entre eles, permitindo seu uso em dado instante. O reuso é determinado sem a necessidade de re-escrita da especificação.

- A definição de uma hierarquia de classes simplifica a extensão da especificação. Conforme exemplificamos no Capítulo 6, a criação da especificação de uma nova linguagem, com base em uma anterior é feita pela simples extensão

Como trabalhos futuros identificamos a expansão de algumas funcionalidades de nossa proposta, as quais ressaltamos a seguir:

- Estudo do impacto nas especificações, dos problemas detectados em [DM01], em relação à composição dos módulos. São necessárias maiores pesquisas para determinarmos a importância destes problemas, no caso da nossa abordagem.
- A notação de classes e conceitos de hierarquia, extensão e instanciação não estão formalmente definidos. No escopo desta dissertação apresentamos a semântica das operações sobre as classes de especificação (sub-classes de *State*) e dos métodos definidos pelo usuário (funções e equações semânticas).
- A notação de ações utilizada neste documento é um sub-conjunto da notação padrão de ações definida por [Mou93]. Visamos expandir esta notação para o conjunto completo definido por [Mos92]. Para este fim a semântica da notação deve ser revista usando algum formalismo mais apropriado, como [Mos99], por exemplo.
- A especificação de linguagens imperativas simples não é suficiente para verificarmos a fundo a capacidade e as limitações de nosso formalismo. Pretendemos testar nossa abordagem na especificação formal de linguagens imperativas completas como a linguagem Pascal, representada em [WM93] utilizando semântica de ações, assim como linguagens de outros paradigmas.
- Baseados no trabalho de [Mou93], desejamos aplicar transformações em semântica de ações orientadas a objetos (eliminação de bindings, etc.). Vemos como pesquisa interessante a verificação da consistência da especificação após a simplificação de elementos desnecessários ou redundantes nos objetos.
- A aplicação de transformações aos objetos é parte de uma iniciativa maior que consiste na geração automática de compiladores, com base em especificações semânticas. Em semântica de ações alguns projetos já foram desenvolvidos, tais como o ACTRESS [BMW92], OASIS [Ør94] e ABACO [MMV⁺02].

Este último item de trabalho futuro sumariza nosso principal intuito na elaboração desta pesquisa. A aplicação de nosso formalismo à geração automática de compiladores é uma das motivações principais à sequência deste projeto.

BIBLIOGRAFIA

- [Acz77] Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, chapter C.7, pages 739–782. North-Holland Publishing Company, 1977.
- [Ast91] Egidio Astesiano. Inductive and operational semantics. IFIP State-of-the-Art Report, pages 51–136. IFIP, Springer-Verlag, 1991.
- [ASU88] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1988.
- [BMW92] Deryck F. Brown, Hermano Moura, and David A. Watt. Actress: an action semantics directed compiler generator. In *CC'92, Proc. 4th Int. Conf. on Compiler Construction, Paderborn*, volume 641 of *Lecture Notes in Computer Science*, pages 95–109. Springer Verlag, 1992.
- [Bro96] D. F. Brown. In *Sort Inference in Action Semantics - Ph.D. Thesis*. University of Glasgow, 1996.
- [dLM88] Silvio Romero de Lemos Meira. *Introdução à Programação Funcional*. VI Escola de Computação, 1988.
- [DM01] K. Doh and P. Mosses. Composing programming languages by combining action semantics modules. In *First Workshop on Language Descriptions, Tools and Applications*, 2001.
- [dS92] Fabio Queda Bueno da Silva. In *Correctness proofs of compilers and debuggers: An approach based on Structural Operational Semantics - Ph.D. Thesis*, 1992.
- [Gay02] J. Gayo. Reusable semantic specifications of programming languages. Spain, 2002. Department of Computer Science - University of Oviedo.

- [Gun91] Carl A. Gunter. Forms of semantic specification. *Bulletin of the European Association for Theoretical Computer Science*, (45):98–113, October 1991. The Logic in Computer Science Column, by Yuri Gurevich.
- [Joh87] Thomas Johnsson. *Compiling Lazy Functional Languages*. Ph.d. thesis, Chalmers University of Technology - Goetheborg, Sweden, 1987.
- [Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6, 1964.
- [Mey97] B. Meyer. In *Object-Oriented Software Construction 2nd ed.* Prentice-Hall, 1997.
- [Mit93] Kevin Mitchell. *Language Semantics and Implementation*. Unpublished Draft, 1993.
- [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.
- [MMV⁺02] L. Menezes, H. Moura, M. Vieira, W. Cansanção, F. Lima, and L. Ribeiro. An action semantics integrated development environment. In *First Workshop on Language Descriptions, Tools and Applications*, Genova, Italy, 2002.
- [Mos92] P. Mosses. Action semantics. In *Action Semantics*. Cambridge University Press, 1992.
- [Mos99] P. D. Mosses. A modular sos for action notation. Technical Report BRICS RS-99-56, University of Aarhus, Dep. of Computer Science, 1999.
- [Mou93] H. Moura. In *Action Notation Transformations - Ph.D. Thesis*. University of Glasgow, 1993.
- [Mus96] M. Musicante. In *On the Relational Semantics of Interleaving Constructors - Ph.D. Thesis*. Universidade Federal de Pernambuco, 1996.
- [Ør94] Peter Ørbaek. Oasis: an optimizing action-based compiler generator. In *Proc. of CC'94*, volume 786 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
- [PJ87] Simon L. Peyton-Jones. *The Implementation of Functional Programming*. Prentice Hall International Series in Computer Science. Prentice Hall, 1987.
- [Plo81] Gordon Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN-19, Aarhus University, Computer Science Department, 1981.

- [Rum94] J. Rumbaugh. In *Object-Oriented Modeling and design*. Campus, 1994.
- [Sch86] D. A. Schmidt. In *Denotational Semantics: A Methodology for Language Development*. Allyn Bacon, 1986.
- [SS71] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In *Proceedings of Symposium on Computers and Automata*, pages 19–46, New York, United States, 1971. Polytechnic Institute of Brooklyn Press.
- [Wat91] D. Watt. In *Programming Language Syntax and Semantics*. Prentice Hall International (UK), 1991.
- [Wik87] Åke Wikström. *Functional Programming Using Standard ML*. Prentice Hall International Series in Computer Science. Prentice Hall, 1987.
- [Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing Series. MIT Press, 1993.
- [WM93] D. Watt and P. Mosses. Pascal action notation. Computer Science Department, Aarhus University, Department of Computing Science, University of Glasgow, 1993.

APÊNDICE A

SEMÂNTICA DE AÇÕES ORIENTADA A OBJETOS

Apresentamos neste apêndice a Semântica de Ações Orientada a Objetos. A Seção A.1 contém a sintaxe da notação, enquanto que a Seção A.2 apresenta sua semântica.

A.1 Sintaxe da Notação

- (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]]
- (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 =
 $\llbracket \text{Function-Name } "[[\text{syntax-tree } "]" \text{ " = " Action }]\rrbracket$.
- (14) Action =
 $\llbracket \text{"complete"} \rrbracket \mid \llbracket \text{"fail"} \rrbracket \mid \llbracket \text{"unfold"} \rrbracket \mid$
 $\llbracket \text{Action "and" Action} \rrbracket \mid \llbracket \text{Action "and then" Action} \rrbracket \mid$
 $\llbracket \text{"unfolding" Action} \rrbracket \mid \llbracket \text{"give" Yielder} \rrbracket \mid$
 $\llbracket \text{"check" Yielder} \rrbracket \mid \llbracket \text{Action "then" Action} \rrbracket \mid$
 $\llbracket \text{"bind" Yielder "to" Yielder} \rrbracket \mid$
 $\llbracket \text{"furthermore" Action} \rrbracket \mid \llbracket \text{Action "hence" Action} \rrbracket \mid$
 $\llbracket \text{Action "moreover" Action} \rrbracket \mid \llbracket \text{Action "before" Action} \rrbracket \mid$
 $\llbracket \text{"store" Yielder "in" Yielder} \rrbracket \mid \llbracket \text{"deallocate" Yielder} \rrbracket \mid$
 $\llbracket \text{"enact" Yielder} \rrbracket \mid \llbracket \text{Action "else" Action} \rrbracket \mid$
 $\llbracket \text{"recursively" "bind" Yielder "to" Yielder} \rrbracket \mid$
 $\llbracket \text{"allocate a" Sort} \rrbracket$.
- (15) Yielder =
 $\llbracket \text{"the" Sort "\#"}n \rrbracket \mid$
 $\llbracket \text{"the" Sort "bound" "to" k} \rrbracket \mid$
 $\llbracket \text{"the" Sort "stored" "in" Yielder} \rrbracket \mid$
 $\llbracket \text{Yielder "with" Yielder} \rrbracket \mid$
 $\llbracket \text{"closure" Yielder} \rrbracket \mid$
 $\llbracket \text{"abstraction of" Action} \rrbracket$.
- (16) Sort =
 $\llbracket \text{"bindable"} \rrbracket \mid \llbracket \text{"cell"} \rrbracket \mid \llbracket \text{"cell" "[" Sort "]" } \rrbracket \mid$
 $\llbracket \text{"storable"} \rrbracket \mid \llbracket \text{"abstraction"} \rrbracket \mid \llbracket \text{"datum"} \rrbracket \mid$
 $\llbracket \text{"integer"} \rrbracket \mid \llbracket \text{"value"} \rrbracket \mid \llbracket \text{"truth-value"} \rrbracket \mid$
 $\llbracket \text{Sort "[" Sort} \rrbracket$.

A.2 Semântica da Notação

A.2.1 Comportamento das operações sobre a Classe *State*

$$S = B, t, b, s \vdash o \triangleright o', t', b', s'$$

A.2.2 Operações básicas

$$B, t, b, s \vdash \text{complete} \triangleright \text{completed}, \{\}, \{\}, s \quad (\text{A.1})$$

$$B, t, b, s \vdash \text{fail} \triangleright \text{failed}, \{\}, \{\}, s \quad (\text{A.2})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t, b, s_1 \vdash a_2 \triangleright \text{completed}, t_2, b_2, s_2}{B, t, b, s \vdash a_1 \text{ and } a_2 \triangleright \text{completed}, t_1 \oplus t_2, b_1 \oplus b_2, s_2} \quad (\text{A.3})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright o_1, t_1, b_1, s_1 \quad o_1 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ and } a_2 \triangleright o_1, t_1, b_1, s_1} \quad (\text{A.4})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t, b, s_1 \vdash a_2 \triangleright o_2, t_2, b_2, s_2 \quad o_2 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ and } a_2 \triangleright o_2, t_2, b_2, s_2} \quad (\text{A.5})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad \text{mergeable } t_1 t_2 \quad B, t, b, s_1 \vdash a_2 \triangleright \text{completed}, t_2, b_2, s_2 \quad \text{mergeable } b_1 b_2}{B, t, b, s \vdash a_1 \text{ and then } a_2 \triangleright \text{completed}, t_1 \oplus t_2, b_1 \oplus b_2, s_2} \quad (\text{A.6})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright o_1, t_1, b_1, s_1 \quad o_1 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ and then } a_2 \triangleright o_1, t_1, b_1, s_1} \quad (\text{A.7})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t, b, s_1 \vdash a_2 \triangleright o_2, t_2, b_2, s_2 \quad o_2 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ and then } a_2 \triangleright o_2, t_2, b_2, s_2} \quad (\text{A.8})$$

$$\frac{B, t, b, s \vdash a \text{ [unfolding } a/\text{unfold}] \triangleright o', t', b', s'}{B, t, b, s \vdash \text{unfolding } a \triangleright o', t', b', s'} \quad (\text{A.9})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright o_1, t_1, b_1, s_1 \quad o_1 \neq \text{failed}}{B, t, b, s \vdash a_1 \text{ or } a_2 \triangleright o_1, t_1, b_1, s_1} \quad (\text{A.10})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{failed}, \{\}, \{\}, s_1 \quad B, t, b, s_1 \vdash a_2 \triangleright o_2, t_2, b_2, s_2}{B, t, b, s \vdash a_1 \text{ or } a_2 \triangleright o_2, t_2, b_2, s_2} \quad (\text{A.11})$$

A.2.3 Operações funcionais

$$\frac{B, t, b, s \vdash y \triangleright d}{B, t, b, s \vdash \text{give } y \triangleright \text{completed}, \{0 \mapsto d\}, \{\}, s} \quad (\text{A.12})$$

$$\frac{B, t, b, s \vdash y \triangleright \text{true}}{B, t, b, s \vdash \text{check } y \triangleright \text{completed}, \{\}, \{\}, s} \quad (\text{A.13})$$

$$\frac{B, t, b, s \vdash y \triangleright \text{false}}{B, t, b, s \vdash \text{check } y \triangleright \text{failed}, \{\}, \{\}, s} \quad (\text{A.14})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t_1, b, s_1 \vdash a_2 \triangleright \text{completed}, t_2, b_2, s_2 \quad \text{mergeable } b_1 b_2}{B, t, b, s \vdash a_1 \text{ then } a_2 \triangleright \text{completed}, t_2, b_1 \oplus b_2, s_2} \quad (\text{A.15})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright o_1, t_1, b_1, s_1 \quad o_1 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ then } a_2 \triangleright o_1, t_1, b_1, s_1} \quad (\text{A.16})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t_1, b, s_1 \vdash a_2 \triangleright o_2, t_2, b_2, s_2 \quad o_2 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ then } a_2 \triangleright o_2, t_2, b_2, s_2} \quad (\text{A.17})$$

$$\frac{d : r}{B, (n \mapsto d) \cdot t, b, s \vdash \text{the } r \# n \triangleright d} \quad (\text{A.18})$$

A.2.4 Operações declarativas

$$\frac{B, t, b, s \vdash y \triangleright d \quad d : \text{bindable}}{B, t, b, s \vdash \text{bind } k \text{ to } y \triangleright \text{completed}, \{\}, \{k \mapsto d\}, s} \quad (\text{A.19})$$

$$\frac{B, t, b, s \vdash a \triangleright \text{completed}, t', b', s'}{B, t, b, s \vdash \text{furthermore } a \triangleright \text{completed}, t', b' \odot b, s'} \quad (\text{A.20})$$

$$\frac{B, t, b, s \vdash a \triangleright o', t', b', s' \quad o' \neq \text{completed}}{B, t, b, s \vdash \text{furthermore } a \triangleright o', t', b', s'} \quad (\text{A.21})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t, b_1, s_1 \vdash a_2 \triangleright \text{completed}, t_2, b_2, s_2 \quad \text{mergeable } t_1 t_2}{B, t, b, s \vdash a_1 \text{ hence } a_2 \triangleright \text{completed}, t_1 \oplus t_2, b_2, s_2} \quad (\text{A.22})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright o_1, t_1, b_1, s_1 \quad o_1 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ hence } a_2 \triangleright o_1, t_1, b_1, s_1} \quad (\text{A.23})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t, b_1, s_1 \vdash a_2 \triangleright o_2, t_2, b_2, s_2 \quad o_2 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ hence } a_2 \triangleright o_2, t_2, b_2, s_2} \quad (\text{A.24})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t, b, s_1 \vdash a_2 \triangleright \text{completed}, t_2, b_2, s_2 \quad \text{mergeable } t_1 t_2}{B, t, b, s \vdash a_1 \text{ moreover } a_2 \triangleright \text{completed}, t_1 \oplus t_2, b_2 \odot b_1, s_2} \quad (\text{A.25})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright o_1, t_1, b_1, s_1 \quad o_1 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ moreover } a_2 \triangleright o_1, t_1, b_1, s_1} \quad (\text{A.26})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t, b, s_1 \vdash a_2 \triangleright o_2, t_2, b_2, s_2 \quad o_2 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ moreover } a_2 \triangleright o_2, t_2, b_2, s_2} \quad (\text{A.27})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad B, t, b_1 \odot b, s_1 \vdash a_2 \triangleright \text{completed}, t_2, b_2, s_2 \quad \text{mergeable } t_1 t_2}{B, t, b, s \vdash a_1 \text{ before } a_2 \triangleright \text{completed}, t_1 \oplus t_2, b_2 \odot b_1, s_2} \quad (\text{A.28})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright o_1, t_1, b_1, s_1 \quad o_1 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ before } a_2 \triangleright o_1, t_1, b_1, s_1} \quad (\text{A.29})$$

$$\frac{B, t, b, s \vdash a_1 \triangleright \text{completed}, t_1, b_1, s_1 \quad t, b_1 \odot b, s_1 \vdash a_2 \triangleright o_2, t_2, b_2, s_2 \quad o_2 \neq \text{completed}}{B, t, b, s \vdash a_1 \text{ before } a_2 \triangleright o_2, t_2, b_2, s_2} \quad (\text{A.30})$$

$$\frac{d : r}{B, t, (k \mapsto d) \cdot b, s \vdash \text{the } r \text{ bound to } k \triangleright d} \quad (\text{A.31})$$

A.2.5 Operações imperativas

$$\frac{B, t, b, s \vdash y_1 \triangleright d \quad B, t, b, s \vdash y_2 \triangleright c \quad s = (c \mapsto d_0) \cdot s' \quad c : \text{cell } [r] \quad d : r}{B, t, b, s \vdash \text{store } y_1 \text{ in } y_2 \triangleright \text{completed}, \{\}, \{\}, \{c \mapsto d\} \odot s} \quad (\text{A.32})$$

$$\frac{B, t, b, s \vdash y \triangleright c \quad s = (c \mapsto d) \cdot s'}{B, t, b, s \vdash \text{deallocate } y \triangleright \text{completed}, \{\}, \{\}, s} \quad (\text{A.33})$$

$$\frac{B, t, b, s \vdash y \triangleright c \quad s = (c \mapsto d) \cdot s' \quad d : r}{B, t, b, s \vdash \text{the } r \text{ stored in } y \triangleright d} \quad (\text{A.34})$$

A.2.6 Operações reflexivas

$$\frac{B, t, b, s \vdash y \triangleright \text{abstraction}(a, t_0, b_0) \quad t_0, b_0, s \vdash a \triangleright o', t', b', s'}{B, t, b, s \vdash \text{enact } y \triangleright o', t', b', s'} \quad (\text{A.35})$$

$$B, t, b, s \vdash \text{abstraction } a \triangleright \text{abstraction}(a, \{\}, \{\}) \quad (\text{A.36})$$

$$\frac{B, t, b, s \vdash y_1 \triangleright \text{abstraction}(a, \{\}, b_0) \quad B, t, b, s \vdash y_2 \triangleright d}{B, t, b, s \vdash y_1 \text{ with } y_2 \triangleright \text{abstraction}(a, \{0 \mapsto d\}, b_0)} \quad (\text{A.37})$$

$$\frac{B, t, b, s \vdash y_1 \triangleright \text{abstraction}(a, t_0, b_0) \quad B, t, b, s \vdash y_2 \triangleright d \quad t_0 \neq \{\}}{B, t, b, s \vdash y_1 \text{ with } y_2 \triangleright \text{abstraction}(a, t_0, b_0)} \quad (\text{A.38})$$

$$\frac{B, t, b, s \vdash y \triangleright \text{abstraction}(a, t_0, \{\})}{B, t, b, s \vdash \text{closure } y \triangleright \text{abstraction}(a, t_0, b)} \quad (\text{A.39})$$

$$\frac{B, t, b, s \vdash y \triangleright \text{abstraction}(a, t_0, b_0) \quad b_0 \neq \{\}}{B, t, b, s \vdash \text{closure } y \triangleright \text{abstraction}(a, t_0, b_0)} \quad (\text{A.40})$$

A.2.7 Operações híbridas

$$\frac{B, \{\}, b, s \vdash a_1 \triangleright o_1, t_1, b_1, s_1}{(0 \mapsto \text{true}) \cdot t, b, s \vdash a_1 \text{ else } a_2 \triangleright o_1, t_1, b_1, s_1} \quad (\text{A.41})$$

$$\frac{B, \{\}, b, s \vdash a_2 \triangleright o_2, t_2, b_2, s_2}{B, (0 \mapsto \text{false}) \cdot t, b, s \vdash a_1 \text{ else } a_2 \triangleright o_2, t_2, b_2, s_2} \quad (\text{A.42})$$

$$\frac{B, t, \{k \mapsto d\} \odot b, s \vdash y \triangleright d \quad d : \text{abstraction}}{B, t, b, s \vdash \text{recursively bind } k \text{ to } y \triangleright \text{completed}, \{\}, \{k \mapsto d\}, s} \quad (\text{A.43})$$

$$\frac{c : r \leq \text{cell} \quad c \notin \text{dom } s}{B, t, b, s \vdash \text{allocate a } r \triangleright \text{completed}, \{0 \mapsto c\}, \{\}, \{c \mapsto \text{uninitialized}\} \oplus s} \quad (\text{A.44})$$

A.2.8 Ambiente para definição de métodos

$$\frac{(f[H]) \in \text{dom}(B), \quad B, t, b, s \vdash A[H/h] \triangleright o, t', b', s'}{B, t, b, s \vdash f[H] \triangleright o, t', b', s'} \quad (\text{A.45})$$

$$\frac{}{B \vdash f[H] = A \triangleright B[f[H] \rightarrow A]} \quad (\text{A.46})$$

APÊNDICE B

SEMÂNTICA DE AÇÕES ORIENTADA A OBJETOS DA LINGUAGEM PURPLE

Neste apêndice apresentamos a Semântica de Ações Orientada a Objetos de nosso estudo de caso - A Linguagem PURPLE. Na Seção B.1 apresentamos sua sintaxe abstrata e na Seção B.2 sua semântica.

B.1 Sintaxe Abstrata da Linguagem PURPLE

- (1) Program =
 [["declare" Declaration "used-in" Command]].
- (2) Declaration =
 [["var" Identifier ":" Type < "=" Expression >?]] |
 [["const" Identifier ":" Type "=" Expression]].
- (3) Type =
 "boolean" | "integer"
- (4) Command =
 [["declare" Declaration "used-in" Command]] |
 [[Identifier "!=" Expression]] |
 [["if" Expression "then" Command < "else" Command >? "end-if"]] |
 [["repeat" Command "until" Expression]] |
 [[Command ";" Command]].
- (5) Expression =
 [["true"]] |
 [["false"]] |
 [[Numeral]] |
 [[Identifier]] |
 [[Expression "+" Expression]] |
 [[Expression "-" Expression]] |
 [[Expression "*" Expression]] |
 [[Expression "<" Expression]] |
 [[Expression "=" Expression]].
- (6) Identifier =
 [[Letter < Letter | Digit >*]].

(7) Numeral =
 [[Digit (Digit)*]].

B.2 Semântica da Linguagem PURPLE

B.2.1 Entidades Semânticas

```
value = integer | truth-value.  
token = value.  
bindable = value | cell.  
cell = cell [ truth-value ] | cell [ integer ].  
storable = value.
```

B.2.2 Identificadores, Numerais e Tipos

```
Class Identifier  
  syntax:  
    Id ::= letter [ letter | digit ]*  
End Class  
  
Class Numeral  
  syntax:  
    N ::= digit+  
  semantics:  
    valuation _ : N -> integer  
End Class  
  
Class Type  
  syntax:  
    T ::= "boolean" | "integer"  
  semantics:  
    allocate-for-type _ : T -> Action  
    allocate-for-type[[ "boolean" ]] =  
      allocate a cell [ truth-value ]  
    allocate-for-type[[ "integer" ]] =  
      allocate a cell [ integer ]  
End Class
```

B.2.3 Declarações

```
Class Declaration
  using I : Identifier, E:Expression
  syntax:
    Dec
  semantics:
    elaborate _ : Dec -> Action
End Class

Class ConstantDec
  extending Declaration
  syntax:
    Dec ::= "const" I ":" T "=" E
  semantics:
    elaborate[["const" I ":" T "=" E]]=
      evaluate E then
        bind I to the value
End Class

Class VariableDec
  extending Declaration
  using T:Type
  syntax:
    Dec ::= "var" I ":" T ["=" E]
  semantics:
    elaborate[["var" I ":" T]]=
      allocate-for-type T
    then
      bind token I to the cell#0
    elaborate[["var" I ":" T "=" E]]=
      evaluate E
      and
        allocate-for-type T
    then
      bind I to the cell#1
      and
        store the value #0 in the cell#1
End Class
```

B.2.4 Comandos

```
Class Command
  syntax:
    Com
  semantics:
    execute _ : Com -> Action
End Class

Class DecCommand
  extending Command
  using D:Declaration, C:Command
  syntax:
    Com ::= "declare" D "used-in" C
  semantics:
    execute[["declare" D "used-in" C]]=
      furthermore elaborate D
      hence
      execute C
End Class

Class Assignment
  extending Command
  using I:Identifier, E:Expression
  syntax:
    Com ::= I "!=" E
  semantics:
    execute[[I "!=" E]]=
      evaluate E
      then
      store the value in the cell bound to I
End Class

Class Selection
  extending Command
  using E:Expression, C1:Command, C2:Command
  syntax:
    Com ::= "if" E "then" C1 [ "else" C2 ] "end-if"
  semantics:
    execute[[ "if" E "then" C1 "end-if" ]]=
      evaluate E
      then
      execute C1
      else
      complete
    execute[[ "if" E "then" C1 "else" C2 "end-if" ]]=
      evaluate E
      then
      execute C1
      else
      execute C2
End Class
```

```

Class Repeat
  extending Command
  using E:Expression, C:Command
  syntax:
    Com ::= "repeat" C "until" E
  semantics:
    execute[[ "repeat" C "until" E ]]=
      unfolding
        execute C
        and then
          evaluate E
          then
            complete
            else
              unfold
End Class

Class Sequencing
  extending Command
  using C1:Command, C2:Command
  syntax:
    Com ::= C1 ";" C2
  semantics:
    execute[[ C1 ";" C2 ]]=
      execute C1 and then execute C2
End Class

```

B.2.5 Expressões

```
Class Expression
  syntax:
    Exp
  semantics:
    evaluate _ : Exp -> Action
End Class

Class SumExp
  extending Expression
  using E1:Expression, E2:Expression
  syntax:
    Exp ::= E1 "+" E2
  semantics:
    evaluate[[ E1 "+" E2 ]] =
      evaluate E1
      and
      evaluate E2
    then
      give sum(the integer#0, the integer#1)
End Class

Class SubtExp
  extending Expression
  using E1:Expression, E2:Expression
  syntax:
    Exp ::= E1 "-" E2
  semantics:
    evaluate[[ E1 "-" E2 ]] =
      evaluate E1
      and
      evaluate E2
    then
      give subtraction (the integer#0, the integer#1)
End Class

Class ProductExp
  extending Expression
  using E1:Expression, E2:Expression
  syntax:
    Exp ::= E1 "*" E2
  semantics:
    evaluate[[ E1 "*" E2 ]] =
      evaluate E1
      and
      evaluate E2
    then
      give product(the integer#0, the integer#1)
End Class
```

```

Class NumeralExp
    extending Expression
    using N:Numeral
    syntax:
        Exp ::= N
    semantics:
        evaluate [[ N ]] =
            give valuation N
End Class

Class IdentifierExp
    extending Expression
    using I:Identifier
    syntax:
        Exp ::= I
    semantics:
        evaluate [[ I ]] =
            give the value bound to I
        or
            give the value stored in the cell bound to I
End Class

Class TrueExp
    extending Expression
    syntax:
        Exp ::= "true"
    semantics:
        evaluate[[ "true" ]] =
            give true
End Class

Class FalseExp
    extending Expression
    syntax:
        Exp ::= "false"
    semantics:
        evaluate[[ "false" ]] =
            give false
End Class

Class LessThan
    extending Expression
    using E1:Expression, E2:Expression
    syntax:
        Exp ::= E1 "<" E2
    semantics:
        evaluate[[ E1 "<" E2 ]] =
            evaluate E1
            and
            evaluate E2
        then
            give less(the integer#0, the integer#1)
End Class

```

```

Class Equality
  extending Expression
  using E1:Expression, E2:Expression
  syntax:
    Exp ::= E1 "=" E2
  semantics:
    evaluate[[ E1 "=" E2 ]] =
      evaluate E1
      and
      evaluate E2
  then
    give equal(the integer#0, the integer#1)
End Class

```

B.2.6 Linguagem

```

Class PurpleLanguage
  using D:Declaration, C:Command
  syntax:
    Prog ::= "declare" D "used-in" C
  semantics:
    run _ : Prog -> Action
    run[[ "declare" D "used-in" C ]]
      elaborate D
    hence
      execute C
End Class

```