Universidade Federal de Pernambuco
Centro de Ciências Exatas e da Natureza
Departamento de Informática

Dissertação de Mestrado

Uso de Orientação a Objetos na Prototipação de Semântica de Ações

por

Luis Carlos de Sousa Menezes

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE CIÊNCIAS EXATAS E DA NATUREZA
DEPARTAMENTO DE INFORMÁTICA

Luis Carlos de Sousa Menezes

Uso de Orientação a Objetos na Prototipação de Semântica de Ações

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Ciências Exatas e da Natureza da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: Hermano Perrelli de Moura

Recife, 13 de Agosto de 1998

Resumo

Semântica de ações é um formalismo utilizado para descrever a semântica de linguagens de programação. Sua estrutura modular e sua notação formal, baseada em termos da língua inglesa, facilitam a especificação de linguagens de programação reais.

A notação formal utilizada por semântica de ações baseia-se em um modelo de especificação denominado álgebras unificadas. Esse modelo define entidades denominadas sortes e operações que manipulam estes sortes.

Nessa dissertação procuramos obter uma representação da notação de ações a partir de suas bases algébricas. Para isso, propomos um método de tradução de especificações algébricas em bibliotecas orientadas a objetos. A escolha de linguagens orientadas a objetos foi motivada pelas similaridades encontradas entre os conceitos existentes em álgebras unificadas e os encontrados no paradigma orientado a objetos.

Uma vez definido o método de tradução, este pode ser utilizado para produzir uma implementação da notação de ações e especificações de linguagens de programação. A principal vantagem dessa abordagem, em comparação com implementações estáticas da notação de ações, é a obtenção de uma representação para semântica de ações que se adeque às particularidades de cada linguagem de programação.

Também propomos nessa dissertação um sistema gerador de compiladores baseado em semântica de ações que utiliza a implementação da especificação de uma linguagem programação, gerada automaticamente pelo tradutor de especificações, para produzir compiladores mais flexíveis.

Abstract

Action semantics is a useful formalism to describe the semantics of programming languages. Its modular structure and formal notation, based on terms of the English language, eases the writing of real programming languages' descriptions.

Action semantics formal notation is based on a specification model known as unified algebras. This model defines entities called sorts and operations that manipulate them.

In this work we intend to give an object-oriented representation for actions semantics. In order to achieve it we propose a translation method that accepts algebraic specifications and yields object-oriented libraries that implement these specifications. The choice of object-oriented languages as target is based on the similarities between the existing concepts in both models.

The translation method will be used to produce an implementation of the action notation and programming language specifications. The main advantage of this approach, when compared to static representations of the action notation, is to obtain a dynamic representation of action semantics that fits for the peculiarities of different programming languages.

We also show how that implementation can be used to produce a compiler prototype for specified languages using a compiler generation system based on action semantics, also proposed in this thesis.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Neste Capítulo forneceremos uma visão geral das idéias que serão mostradas nessa dissertação. Inicialmente descreveremos os principais motivos que levaram a definição do sistema gerador de compiladores ABACO, descrito nesta tese. A seguir descreveremos os principais módulos do sistema ABACO. Finalmente mostraremos como esses tópicos estão dispostos nesse texto.

1.1 Motivação

Semântica de ações [Mos92, Wat91] é um formalismo utilizado para a descrição da semântica de linguagens de programação, sua principal vantagem é a facilidade de elaboração de especificações de linguagens reais e a legibilidade dessas especificações.

Uma aplicação importante de semântica de ações é o seu uso para geração automática de compiladores. Nessa área foram propostos vários métodos de análise e otimização que permitem a geração de compiladores comparáveis aos escritos manualmente. Alguns desses métodos são descritos em [Ørb93, Pal92, BMW92]. Esses métodos são definidos utilizando um subconjunto estático para semântica de ações o que dificulta a inserção de novas propriedades necessárias para a definição de linguagens de programação reais.

Nessa dissertação apresentamos um método para geração de compiladores para semântica de ações baseado na notação formal usada para definir as características do formalismo. A abordagem mostrada nessa dissertação possui como principal vantagem o fato do mesmo não se limitar, necessariamente, à um subconjunto pré-definido da notação de ações. Essa característica simplifica o processo de adição de novos tipos de dados ou controle e permite a construção de compiladores que representem de forma mais correta as características da linguagem especificada.

1.2 O Método Proposto

O sistema ABACO, sistema gerador de compiladores para semântica de ações, que será proposto nesta dissertação é formado pelos seguintes módulos:

1. Um algoritmo para tradução de especificações algébricas unificadas, notação formal utilizada por semântica de ações, em código orientado a objetos.

2. A descrição de semântica de ações através de sua notação formal, que será utilizada para produzir implementações automáticas para esse formalismo.

3. Um algoritmo gerador de código para a notação de ações.

1.3 Organização

Essa tese esta organizada em duas grandes partes que passamos a descrever abaixo.

A primeira parte da tese tem como tema principal o modelo de álgebras unificadas e o método de tradução orientada a objetos de especificações algébricas. Durante essa parte descrevemos as principais características do modelo (Capítulo 2) e propomos o método de tradução de especificações algébricas unificadas, capaz de produzir um código orientado a objetos equivalente a uma dada especificação (Capítulo 3).

A segunda parte dessa dissertação aborda o formalismo de semântica de ações propriamente dito. Nessa parte descrevemos as características de semântica de ações (Capítulo 4), comentamos alguns trabalhos em geração de compiladores baseada em semântica de ações (Capítulo 5), e descrevemos como o método de tradução de especificações algébricas pode ser utilizado para produzir uma interpretação da especificação de uma linguagem de programação, que poderá ser utilizada para aperfeiçoar os métodos geradores de compiladores propostos na literatura (Capítulo 5 e 6).

Finalizando essa dissertação, mostramos as contribuições produzidas pela definição do método proposto e indicamos alguns possíveis tópicos de estudos futuros (Capítulo 7).

Capítulo 2

Especificações Algébricas

Nesse capítulo descreveremos o modelo de álgebras unificadas, um modelo de especificações algébricas sortidas ordenadas que forma a base matemática para definição de semântica de ações.

Na primeira parte desse capítulo (seção 1), descreveremos as principais características de especificações algébricas sortidas ordenas. Essa parte foi baseada principalmente nos conceitos descritos em [Gogar, GM92]

Na segunda parte desse capítulo (seções 2 e 3), descreveremos a sintaxe de uma especificação algébrica unificada e as particularidades existentes nesse modelo.

2.1 Algébricas Sortidas Ordenadas

Especificações algébricas sortidas ordenadas (Order Sorted Algebras-OSA) definem um conjunto de entidades denominadas de sortes, que são formadas por conjuntos de indivíduos que possuem propriedades comuns.

Os sortes de uma OSA estão relacionados por meios de inclusão de sortes. Um sorte a inclui um sorte b se todos os indivíduos pertencentes ao sorte b pertencerem também ao sorte a. Nesses casos dizemos de o sorte a é um supersorte do sorte b ou que b é um subsorte do sorte a. A relação de inclusão de sortes é útil para a modelagem de heranças e subtipos.

O comportamento dinâmico de uma especificação é definida pelos operadores desta. Um operador recebe como argumentos indivíduos pertencentes a um conjunto determinado sortes e produz como resultado um indivíduo de um novo sorte.

Devido as suas características, especificações algébricas sortidas e ordenadas se mostram úteis na definição de tiops abstratos de dados e especificação de sistemas.

2.1.1 Assinaturas

Uma assinatura de uma especificação algébrica indica os sortes e operadores definidos pela especificação. O formato de uma assinatura é dada pela tupla: $(S, \leq, \Sigma)$, onde S é o conjunto de sortes definidos pela álgebra, $\leq$ é uma relação de ordem parcial e define a relação de inclusão existente entre os sortes definidos em S e Σ é o conjunto dos símbolos operacionais (operações) definidos pela assinatura.

O conjunto de símbolos operacionais Σ é definido através da expressão:
$\{\Sigma_{x,y} | x \in S^*, y \in S\}$, onde $\Sigma_{x,y}$ indica as operações que recebem os argumentos dos sortes x e produzem elementos do sorte y. Se σ é uma operação da assinatura então:

- Se $\sigma \in \Sigma_{x,y}$, então σ possui aridade x e sorte y.
- Se $\sigma \in \Sigma_{[],y}$, então σ é uma constante de sorte y.

2.1.2 Álgebras

Uma álgebra A possui uma assinatura $(s, \leq, \Sigma)$, se:

- Para todo sorte $x \in s$ existir um conjunto relacionado M_x em A.
- Para todos os sortes x_1, x_2 definidos pela assinatura: $x_1 \leq x_2 \;\Rightarrow\; M_{x_1} \subset M_{x_2}$.
- Para toda operação constante $\lambda \in \Sigma_{[],x}$, deve existir um valor $v(\lambda) \in M_x$.

- Para todo símbolo funcional $\lambda \in \Sigma_{x,y}$, onde $x = (x_1, x_2, ..., x_n)$, deve existir em A uma operação M_λ que receba como parâmetros valores dos conjuntos M_{x_1}, M_{x_2}, ..., M_{x_n} e produza valores do conjunto M_y.

2.1.3 Termos

O conjunto de termos válidos de uma especificação de assinatura $(s, \leq, \Sigma)$ é definido pelas seguintes regras:

1. Toda constante $\sigma \in \Sigma_{[],x}$ é um termo de sorte x válido na especificação.
2. Se t_1, t_2,...,t_n são termos de sortes $x_1, x_2, ..., x_n$ respectivamente e $\sigma \in \Sigma_{(x_1,x_2,...,x_n),y}$, então $\sigma(t_1, t_2, ..., t_n)$ é um termo válido da especificação e possui sorte y.

O conjunto de termos válidos descrito acima pode ainda ser extendido pela adição de símbolos variáveis que podem representar qualquer termo válido da especificação. A expressão $var(t)$ indica o conjunto de variáveis existente no termo t.

As variáveis de um termo podem ser substituídas por termos constantes da especificação. A seguinte notação:

$$t(x_1 \leftarrow t_1, x_2 \leftarrow t_2, ..., x_n \leftarrow t_n)$$

representa o termo t com as variáveis $(x_1, x_2, ..., x_n)$ substituídas pelos termos $(t_1, t_2, ..., t_n)$ respectivamente.

2.1.4 Equações

Os termos válidos em uma assinatura Σ estão associados a valores pertencentes a uma algebra, sendo que, um mesmo valor pode ser produzido a partir de vários termos diferentes. As equações de uma especificação algébrica são utilizadas para indicar a igualdade entre dois termos diferentes da especificação.

Uma equação em uma especificação algébrica é formada pela tripla (X, T_e, T_d), onde: X é o conjunto de variáveis da equação e os termos T_e e T_d estabelecem a igualdade entre os termos distintos. Essa equação pode ser expressa usando-se uma notação similar a mostrada abaixo:

- $(\forall X)\ T_e = T_d$

Um outro tipo de equação são as equações condicionais, definidas pela tupla (X, T_e, T_d, C), onde: X é o conjunto de variáveis da equação, C é uma condição que a equação deve satisfazer e os termos T_e e T_d estabelecem a igualdade entre termos distintos. Uma equação condicional pode ser expressa de forma mais legível usando-se uma notação como:

- $(\forall X)\ T_e = T_d$ **if** C

2.1.5 Dedução em Álgebras

A igualdade entre termos distintos pode ser inferida utilizando-se as seguintes regras:

1. (**Reflexividade**) Toda equação da forma:

   $(\forall X)\ t = t$

   é valida.

2. (**Simetria**) Se a equação:

   $(\forall X)\ x = y$

   é valida, então a seguinte equação também será válida:

   $(\forall X)\ y = x$

3. (**Transitividade**) Se as equações:

   $(\forall X)\ x = y$

$(\forall X)\ y = z$

são validas, então a seguinte equação também será válida:

$(\forall X)\ x = z$

4. (**Congruência**) Se t é um termo da especificação e x uma variável de t e a equação:

   $(\forall X)\ t_1 = t_2$

   é válida. Então a sequinte equação é válida:

   $(\forall X)\ t(x \leftarrow t_1) = t(x \leftarrow t_2)$

5. (**Substitutibilidade**) Se a equação:

   $(\forall X)\ t_1 = t_2$

   é válida. Se x é uma variável de t_1 e t_2 e t um termo, então a sequinte equação é válida:

   $(\forall X)\ t_1(x \leftarrow t) = t_2(x \leftarrow t)$

2.1.6 Sistemas de Reescrita de Termos

Sistemas de reescritas de termos (Term Rewriting System - TRS)[DJ90] são especificações algébricas com características particulares. A principal característica desse tipo de especificação é a utilização de um sistema de dedução simplificado que pode ser facilmente descrito computacionalmente, dessa forma esses sistemas podem ser facilmente implementados. Nessa Seção faremos um comentário sobre sistemas de reescritas e mostraremos algumas propriedades úteis para esse tipo de especificações.

2.1.6.1 Características de um TRS

Definição 2.1 *(Equação de reescrita) Uma equação da forma:*

$(\forall X)\ t_e = t_d$

é uma equação de reescrita se $var(t_d) \subseteq V(t_e)$.

De forma análoga, uma equação condicional da forma:

$$(\forall X)\ t_e = t_d \ \textsf{if}\ c$$

é uma equação de reescrita se: $var(t_d) \cup var(c) \subseteq var(t_e)$.

Definição 2.2 *(Sistemas de reescrita) Uma especificação algébrica forma um sistema de reescrita de termos quando suas equações são de reescrita.*

2.1.6.2 Regras de Dedução em TRS

A dedução de igualdade entre equações de reescrita é feita através da aplicação sucessiva de passos de reescrita.

Definição 2.3 *(Passo de reescrita) Um passo de reescrita, representado pelo símbolo ($\Rightarrow$), é uma relação entre termos de uma especificação. Dois termos x e y estão relacionados ($x \Rightarrow y$) se a especificação possuir uma equação:*

$$(\forall X)\ t_e = t_d$$

e $x = t_e(x_1 \leftarrow t_1, ..., x_n \leftarrow t_n)$ e $y = t_d(x_1 \leftarrow t_1, ..., x_n \leftarrow t_n)$ onde: $(x_1, ..., x_n)$ são variáveis pertencentes aos termos t_e e t_d, $(t_1, ..., t_n)$ são termos da especificação. Nesses casos dizemos que x pode ser reescrito para o termo y.

Definição 2.4 *A sequência abaixo:*

$$t_1 \ \Rightarrow \ t_2 \ \Rightarrow \ ... \ \Rightarrow \ t_n$$

define uma sequência de reescritas, se: $t_1 \Rightarrow t_2, t_2 \Rightarrow t_3, ..., t_{n-1} \Rightarrow t_n$.

Definição 2.5 *(Termo Normal) Um termo t é normal (ou reduzido), se: ele não puder ser reescrito para nenhum outro termo.*

Definição 2.6 *(Especificação terminante) Um sistema de reescrita é terminante, se ela não possuir nenhuma sequência de reescritas infinita.*

Definição 2.7 *(Especificação confluente) Um sistema de reescrita é confluente se todos os termos da especificação admitirem apenas uma única forma normal.*

2.2 Especificações Algébricas Unificadas

O modelo de especificação algébrica utilizado por semântica de ações para descrever linguagens de programação é denominado de modelo de álgebras unificadas, esse modelo está definido detalhadamente em [Mos89].

O modelo de álgebras unificadas é utilizado para definir álgebras sortidas e ordenadas. Esse modelo de especificação possui algumas características que o distinge de outros modelos de especificações, essas características são mostradas abaixo:

- O modelo unificado não faz distinção entre os conceitos de valores e sortes. Em uma especificação algébrica unificada os valores (ou indivíduos) são interpretados como sortes unitários. Da mesma forma, os sortes são considerados valores que podem ser passados como parâmetros de operadores ou produzidos como resultados destes.
- A relação de inclusão de sortes é definida através dos axiomas da especificação.
- As operações que agem sobre sortes devem ser monotônicas isso significa que se s é um subsorte de s' e op é uma operação, então o resultado da operação $op(s)$ deve ser um subsorte de $op(s')$.
- Existem duas operações pré-definidas pelo modelo, responsáveis pela manipulação de sortes:
    1. União de sortes: essa operação recebe dois sortes como parâmetros e devolve um novo sorte formado pela união dos valores pertencentes aos dois sortes. Por

exemplo, o sorte dos números não nulos é formado pela união dos sortes dos números positivos e dos números negativos.

2. Interseção de sortes: essa operação recebe dois sortes como parâmetros e devolve um novo sorte formado pelos elementos comuns aos dois sortes fornecidos. Por exemplo, o sorte formado pelo número 2 pode ser obtido pela interseção dos sortes dos números primos e dos número pares.

- O sorte de nome nothing é pré-definido pelo modelo e denota um sorte que não possui nenhum elemento e que é subsorte de todos os sortes.

2.2.1 Notação

A notação usada para definir uma especificação é descrita detalhadamente em [Mos89]. A sintaxe básica de uma especificação é mostrada na Figura 2.1. Nessa notação uma especificação algébrica[1] é dividida em duas partes: assinatura e axiomas.

2.2.1.1 Assinatura

A assinatura de uma especificação é definida através das seguintes sentenças:

- **introduces:** S_1 , S_2, f_1_, _ f_2 _, ...

  Indicam os sortes S_1, S_2, ... e os operadores f_1 _ , _ f_2 _,... que serão definidos pela especificação. Os operadores são distinguidos dos sortes através do símbolo "_" que indica o posicionamento dos operandos.

- **needs:** $espec_1$, $espec_2$, ...

  Indica que a especificação usará os elementos definidos pelas especificações $espec_1$, $espec_2$, etc.

[1] A partir desse ponto utilizaremos o termo especificações algébricas para identificar especificações algébricas unificadas.

(1) Especificação = Assinatura* Axioma*.

(2) Assinatura = "**introduces:**" nome ⟨ "," nome ⟩* "." |
"**needs:**" nome ⟨ "," nome ⟩* "." |
"**includes:**" nome ⟨ "," nome ⟩* "." |
Funcionalidade.

(3) Funcionalidade = nome ": :" SorteExpressão ⟨ "," SorteExpressão ⟩* "→ " SorteExpressão ⟨ "(" ⟨ Atributo "," ⟩* Atributo ")" ⟩$^?$.

(4) Atributo = "*total*" | "*partial*" | "*restricted*" |
"*strict*" | "*linear*" | "*injective*" |
"*associative*" | "*commutative*" | "*idempotent*" |
"*unit*" "*is*" SorteExpressão.

(5) Axioma = ⟨ Formula " ⇒ " ⟩* Formula.

(6) Formula = Expressão ":" Expressão |
Expressão "=" Expressão |
Expressão "≤" Expressão |
Expressão "≥" Expressão.

(7) Expressão = Sorte |
Expressão " | " Expressão |
Expressão " & " Expressão.

(8) Sorte = nome | *nome* | nome "(" ⟨ Expressão "," ⟩* Expressão ")".

(9) nome = string.

Figura 2.1: Sintaxe de uma Especificação Algébrica Unificada.

- **includes:** $espec_1$, $espec_2$, ...

  Indica que a especificação, assim como todas as especificações que usem os elementos dessa especificação, usarão os elementos definidos pelas especificações $espec_1, espec_2$, etc.

- o _ _ _ ... :: S_0 , S_1 ... $S_{n-1} \rightarrow S_n$ (p)

  Essa sentença define a funcionalidade do operador o, estabelecendo que ele recebe como parâmetros elementos dos sortes S_0,..., S_{n-1} e devolve como resultado um elemento do sorte S_n.

  O símbolo p define alguns atributos da operação o. Uma operação pode receber um ou mais dos seguintes atributos:

  - *total*: indica que a operador retornará um indivíduo do sorte S_n quando todos os parâmetros forem indivíduos dos seus sortes. Além disso indica que o operador é *strict* e *linear*.
  - *partial*: o operador pode retornar o valor nothing indicando que não está definido para determinados parâmetros.
  - *restricted*: indica que o operador não pode ser redefinido com argumentos e resultados diferentes dos especificados.
  - *strict*: o operador retorna o sorte nothing quando um dos parâmetros for nothing
  - *linear*: o operador é distributivo com relação a união de sortes.
  - *injective*: o operador é injetivo.
  - *associative*: o operador é associativo.
  - *commutative*: o operador é comutativo.
  - *idempotent*: o operador é idempotent ou seja: $o(x, x) = x$.
  - *unit is* U: o sorte U é unidade para o operador o.

  Os operadores podem aparecer nas formas: pré-fixada, pós-fixada e mista, a o posicionamento dos argumentos da operação é dado pela ocorrência do símbolo "_" no nome do operador.

2.2.1.2 Axiomas

Essa parte de uma especificação algébrica define a relação de inclusão entre os sortes e o comportamento das operações declaradas na especificação, para isso são utilizadas expressões de sortes na forma de cláusulas de Horn. Cláusulas de Horn, são expressões da forma:

$$(p_1 \wedge p_2 \ldots p_{n-1}) \Rightarrow p_n.$$

essas expressões estabelecem que se as sentenças p_1, ..., p_{n-1} forem verdadeiras, então a sentença p_n será também verdadeira. As sentenças p_x podem assumir uma das seguintes formas:

- $e : S$: É verdadeiro se o sorte e é um indivíduo do sorte S. Um sorte é um indivíduo de outro se ele for um subsorte e se possuir apenas um elemento.
- $S_1 = S_2$: É verdade se os dois sortes são iguais. Dois sortes são iguais se eles possuírem os mesmos elementos.
- $S_1 \leq S_2$: É verdade se o sorte S_1 é um subsorte de S_2.
- $S_1 \geq S_2$: É verdade se o sorte S_1 é um supersorte de S_2.

As convenções sintáticas usadas nessa tese para descrever os sortes são definidas abaixo.

1. Os símbolos da forma e (em letras minúsculas) representam termos definidos pela assinatura.
2. Os símbolos na forma S_n (em letras maiúsculas) representam:
   - Um termo definido pela assinatura.
   - O resultado da aplicação de uma das operações pré-definidas de sortes. A notação usada para representar as operações é dada abaixo:
     - União de sortes: representada pelo operador infixo "|".

- Interseção de sortes: representada pelo operador infixo "&".

- O símbolo especial $\square$ que tem como significado um sorte que ainda vai ser definido.

Álgebras unificadas permitem uma série de abreviações sintáticas que não serão tratadas explicitamente nessa tese. Por exemplo:

1. Se um axioma A possui um termo da forma $o(..., x : y, ...)$, ele será uma abreviação do axioma: $x : y \Rightarrow A'$, onde A' é o axioma A com o termo $o(..., x : y, ...)$ substituído por $o(..., x, ...)$. Por exemplo, o axioma:

   sucessor(i:natural) $= i + 1$.

   é uma abreviação de:

   i:natural $\Rightarrow$ sucessor $i = i + 1$.

2. $i!$ é uma abreviação de $i : i$.

2.3 Exemplos de Especificações Algébricas Unificadas

Essa seção descreverá dois exemplos de especificações algébricas escritas utilizando álgebras unificadas. Esses exemplos serão utilizados no próximo capítulo para ilustrar o método de tradução de especificações algébricas.

Números Naturais

introduces: natural, positive-integer, sucessor _, sum _ _,0, fib _.

(1) sucessor _ :: natural $\rightarrow$ natural (*total*, *injective*).

(2) sum _ _ :: natural, natural $\rightarrow$ natural (*commutative*, *unit is* 0).

(3) fib _ :: natural $\rightarrow$ natural.

(4) 0 : natural.

(5) sum 0 $n = n$.

(6) sum (sucessor m n) = sucessor (sum m n).

(7) fib 0 = 0.

(8) fib sucessor 0 = sucessor 0.

(9) fib sucessor sucessor x = sum (fib sucessor x) (fib x).

Essa especificação define os números naturais, a sua assinatura é formada pela sentença **introduces**, que indica os elementos definidos pela especificação, e pelas declarações das funcionalidades dos operadores sucessor _, sum _ _ e fib _ (sentenças 1, 2 e 3).

O axioma 4 define o sorte 0 como um indivíduo do sorte natural.

Os outros axiomas definem o comportamento dos operadores sum _ _ e fib _.

Pilhas

introduces: stack, empty-stack, item, push _ _, pop _, top _.

needs: Numeros Naturais.

(1) push _ _ :: item , stack $\rightarrow$ stack .

(2) pop _ :: stack $\rightarrow$ stack ($partial$).

(3) top _ :: stack $\rightarrow$ item ($partial$).

(4) item $\geq$ natural.

(5) empty-stack : stack.

(6) top push n $p = n$.

(7) pop push n $p = p$.

(8) top empty-stack = nothing.

(9) pop empty-stack = nothing.

A especificação acima define uma pilha de inteiros, a sua assinatura identifica na sentença **introduces** os sortes e operadores que serão definidos, a sentença **needs** define que essa especificação necessitará dos elementos definidos pela especificação dos números naturais. As sentenças 1, 2 e 3 definem as funcionalidades dos operadores push _ _, pop _ e top _.

As sentenças 4 e 5 da especificação são formadas por axiomas que definem as relações existentes entre os sortes item e natural e entre empty-stack e stack, respectivamente.

As sentenças 6 a 9 da especificação definem o comportamentos dos operadores da especificação.

Capítulo 3

Tradução OO de Especificações Algébricas

Esse capítulo tem como objetivo apresentar um método para geração de código orientado a objetos a partir de especificações algébricas escritas usando-se o modelo de álgebras unificadas apresentado no Capítulo 2.

Nas primeiras seções forneceremos uma visão geral do processo de tradução (Seção 1) e descreveremos o resultado da tradução de uma especificação algébrica (Seção 2).

Após essa introdução apresentaremos o processo de tradução propriamente dito. Inicialmente descreveremos uma simplificação do modelo de especificação de álgebras unificadas (seções 3 e 4) para depois tratarmos do processo de tradução (Seção 5). A seguir indicaremos algumas otimizações que podem ser acrescentadas ao método para aumentar a eficiência do código gerado pelo método de tradução (Seção 6).

Finalizando esse capítulo, faremos uma análise crítica do método de tradução onde discutiremos as limitações do método (Seção 7) e comentaremos os resultados obtidos com uma implementação do método (seções 8 e 9).

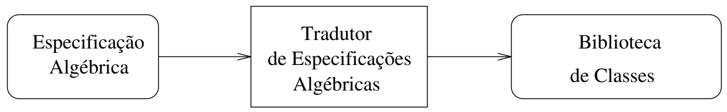

Figura 3.1: Tradutor OO de Especificações Algébricas

3.1 Visão Geral do Método de Tradução

O método descrito nesse capítulo define um tradutor de especificações algébricas cujo funcionamento é mostrado na Figura 3.1. Esse tradutor analisará uma especificação algébrica fornecida e produzirá uma biblioteca de classes que implemente os tipos de dados especificados. As características dessa biblioteca são mostradas na Seção 3.2.

A biblioteca de classes resultante do processo de tradução de uma especificação algébrica (Seção 3.2) é capaz de:

- Representar os termos válidos segundo a especificação fonte.
- Realizar simplificações de termos que puderem ser aplicadas, segundo os axiomas da especificação.
- Reconhecer relações de igualdade e inclusão de sortes entre os termos.

3.2 Resultado da Tradução

O processo de tradução de uma especificação produzirá uma biblioteca de classes, essa biblioteca define um conjunto de classes e alguns objetos padrões que representarão o comportamento da especificação.

A escolha de linguagens orientadas a objetos [Mey97] como resultado do processo de tradução se deve às similaridades encontradas entre os conceitos do paradigma orientado a objetos e os conceitos existentes em especificações algébricas unificadas. Durante a pesquisa realizada para definir o método de tradução, detectamos as seguintes similaridades:

1. Os sortes possuem as seguintes similaridades:

    - Por serem utilizados na definição de tipos de dados abstratos, os sortes são similares ao conceito de classes. Essa similaridade é utilizada principalmente durante a tradução das funcionalidades dos operadores da especificação.
    - Por serem passados como parâmetros de operadores, o conceito de sorte também podem ser considerado similar ao conceito de objeto. Esse similaridade é explorada durante a tradução dos termos de uma especificação.

    Cada uma dessas similaridades serão utilizadas simultaneamente durante o processo de tradução.

2. O conceito de inclusão de sortes em especificações algébricas é similar à relação de herança do paradigma orientado a objetos.

3. As operações definidas por uma especificação algébrica, são similares as métodos existentes em programas orientados a objetos.

4. Os termos gerados pelas operações são descritos como objetos de classes especialmente projetadas para este fim.

Como linguagem alvo do processo de tradução utilizaremos a linguagem C++ [Str91], que foi escolhida devido ao seu uso difundido e por suportar os conceitos de orientação a objetos necessários ao processo.

Uma das principais deficiências de linguagens como C++ é a necessidade do programador definir rotinas para gerenciamento de memória. Essa característica facilita o aparecimento de erros em programas complexos. Por motivos de simplificação da descrição do processo de tradução não consideraremos o problema da desalocação de memória, implementações do método devem ser capazes de adicionar segmentos de código que realizem essa tarefa.

3.3 Especificações Algébricas Unificadas Simplificadas

O modelo de álgebras unificadas permite uma grande variedade de axiomas, isso faz com que uma mesma sentença possa ser expressa por várias expressões diferentes. Um exemplo disso é o axioma que define um sorte a como um subsorte de um outro sorte b, que pode ser expresso das seguintes formas:

(1) $\mathsf{a} \leq \mathsf{b}$.

(2) $\mathsf{b} \geq \mathsf{a}$.

(3) $\mathsf{b} = \mathsf{b} \mid \mathsf{a}$.

(4) $\mathsf{a} = \mathsf{a}\ \&\ \mathsf{b}$.

Levando em conta essa característica, simplificaremos o processo de tradução de especificações algébricas definindo-o apenas para um modelo de especificações simplificado. A sintaxe de uma especificação nesse modelo simplificado, mostrada na Figura 3.2, é formada por um subconjunto da sintaxe das especificações de álgebras unificadas (Figura 2.1). A semântica das expressões nesse modelo é a mesma das expressões em álgebras unificadas vistas no Capítulo 2.

As principais diferenças existentes entre especificações algébricas unificadas e especificações algébricas unificadas simplificadas são listadas abaixo.

1. Especificações simplificadas não permitem a utilização de elementos definidos em outras especificações (sentenças **needs** e **includes**).

2. As declarações de funcionalidades em especificações simplificadas não podem declarar como parâmetros e resultado dos sentença **introduces** da especificação, não sendo permitidos o uso de operadores de sortes para nomes de sortes como parâmetros e resultados.

3. Os únicos tipos de expressões de sortes permitidos são:

(1) Especificação = Assinatura* Axioma*.

(2) Assinatura = "**introduces:**" nome $\langle$ "," nome $\rangle^*$ "." **|**
Funcionalidade .

(3) Funcionalidade = nome ": :" nome $\langle$ "," nome $\rangle^*$ "$\rightarrow$ " nome $\langle$ "(" $\langle$ Atributo "," $\rangle^*$ Atributo ")" $\rangle^?$.

(4) Atributo = "*total*" **|** "*partial*" **|** "*restricted*" **|**
"*strict*" **|** "*linear*" **|** "*injective*" **|**
"*associative*" **|** "*commutative*" **|** "*idempotent*" **|**
"*unit*" "*is*" Sorte.

(5) Axioma = $\langle$ Formula " $\Rightarrow$ " $\rangle^*$ Formula.

(6) Formula= Sorte ":" Sorte **|** Sorte "=" Expressão.

(7) Expressão = Sorte **|**
Sorte $\langle$ " **|** " Sorte $\rangle^+$ **|**
Sorte $\langle$ " & " Sorte $\rangle^+$.

(8) Sorte = nome **|** *nome* **|** nome "(" $\langle$ Expressão "," $\rangle^*$ Expressão ")".

(9) nome = string.

Figura 3.2: Sintaxe de uma Especificação Algébrica Unificada Simplificada.

- igualdade entre sortes, sendo que somente os termos do lado direito podem conter operações de união ou interseção de sortes.
- declaração de elemento.

3.4 Simplificação de Especificações Algébricas Unificadas

Apesar de definirmos a tradução apenas para uma simplificação do modelo de álgebras unificadas, o método descrito nesse capítulo não perde generalidade, pois as especificações unificadas podem ser simplificadas de forma a preencherem os requisitos necessários a uma especificação algébrica simplificada. Essa simplificação é feita através da aplicação exaustiva de um conjunto de regras de conversão que estão definidas nessa seção.

Durante a explicação das regras de conversão utilizaremos as seguintes convenções de nomenclatura:

- O símbolo S_n, onde n é um número natural, representa expressões formadas por termos válidos na especificação ou operações de união (ou intersecção) entre estes.
- O símbolo s_n, onde n é um número natural, representa expressões formadas por termos válidos na especificação.
- O símbolo o_n, onde n é um número natural, representa um operador da especificação.
- O símbolo r_n, onde n é um número natural, representa um novo sorte que deve ser adicionado à especificação.

Regras de Simplificação

A simplificação de uma especificação algébrica unificada é feita seguindo-se duas etapas:

1. Simplificação da Assinatura.
2. Simplificação dos Axiomas.

Etapa 1: Simplificação da Assinaturas

A primeira etapa tem como o objetivo substituir declarações não simplificadas não simplificadas da assinatura por declarações equivalentes simplificadas. Isso é feito através dos seguintes passos:

Passo 1.1: Simplificação da Estrutura de Módulos

Esse passo tem como objetivo a eliminação de sentenças needs e includes da especificação. Para isso todas as importações feitas pela especificação não simplificada devem ser substituídas pelas declarações existentes nas especificações importadas até não haver mais sentenças needs e includes na especificação corrente.

Exemplo

A especificação de pilhas importa a especificação dos números naturais através da sentença:

needs: Numeros Naturais

Para simplificarmos essa especificação, a sentença acima deve ser removida e as assinaturas e axiomas da especificação Números Naturais devem ser inseridos na nova especificação.

□

Passo 1.2: Simplificação das Funcionalidades

Esse passo procura colocar as declarações de funcionalidades conforme o formato do modelo simplificado. Para isso utilizamos as seguintes regras:

Regra 1.2.1

Funcionalidades da forma:

o_1 _ ... :: s_1 , S_i ... $\rightarrow s_n$

Onde: $1 \leq i \leq n$. Então essa funcionalidade deve ser substituída por:

o_1 _ .. :: s_1, ..., r_1, ... $\rightarrow s_n$.

$r_1 = S_i$.

Exemplo

$\_ + \_$ :: string, (string | integer) $\rightarrow$ string.

O axioma acima é inválido na notação simplificada, pois expressões na forma (string | integer) não são válidas dentro de funcionalidades. Para tornar essa declaração válida devemos criar um novo sorte, se ainda não existir um, de valor (string | integer). Assim o segmento de especificação resultante será:

$\_ + \_$ :: string, novo-sorte $\rightarrow$ string.

novo-sorte = string | integer.

$\square$

Regra 1.2.2: Operadores Comutativos

Se uma operador o é comutativo então para todo axioma da forma:

$o(x,y) = e$

da especificação fonte, produz um outro axioma da forma:

$o(y,x) = e$

onde e é uma expressão de sorte.

Regra 1.2.3: Valor Unitário

Se um operador o possui como valor unitário o sorte s então os seguintes axiomas serão inseridos na especificação:

$o(s,z) = z.$

$o(z,s) = z.$

Regra 1.2.4: Operadores Idempotentes

Se um operador o é idempotente então o seguinte axioma é inserido na especificação:

$o(x,x) = x$.

Regra 1.2.5: Operadores Estritos

Se um operador o é estrito então os seguintes axiomas são inseridos na especificação:

o(nothing,x,...) = nothing.

o(x,nothing,...) = nothing.

...

o(...,x,nothing) = nothing.

Exemplo

Se em uma especificação de pilhas, a operação sum _ _ possui como valor unitário o sorte 0, devemos inserir na especificação os seguintes axiomas:

sum 0 $x = x$.

sum x 0 $= x$.

□

Etapa 2: Simplificação das Relações

Essas regras tem como objetivo converter os axiomas para uma forma válida de acordo com o modelo simplificado. Isso é feito através das seguintes regras:

Regra 2.1

Axiomas da forma

S_1 Rel S_2.

onde Rel é uma das relações entre sortes ($\leq$, $\geq$, $=$,:). são simplificadas da seguinte forma:

$r = S_1$.

r Rel S_2.

Exemplo

O axioma:

(natural | negative) = (zero | nozero-integer).

pode ser traduzido para a notação simplificada como:

integer = natural | negative.

integer = zero | nozero-integer.

$\square$

Regra 2.2

Axiomas da forma

(1) $s_1 = S_2$ | S_3 | ... | S_n | $\square$.

(2) $s_1 \geq S_2$ | S_3 | ... | S_n | $\square$.

(3) $s_1 \leq S_2$ | S_3 | ... | S_n | $\square$.

(4) $s_1 = \square$.

são simplificadas da seguinte forma:

(1) $s_1 \geq S_2$ | S_3 | ... | S_n.

(2) $s_1 \geq S_2$ | S_3 | ... | S_n.

(3) $s_1 = s_1$.

(4) $s_1 = s_1$.

Exemplo

O axioma:

bindable $\geq \Box$.

é convertido para o modelo simplificado como:

bindable = bindable.

$\Box$

Regra 2.3

Axiomas da forma

(1) $s_1 \leq S_2$.

(2) $s_1 \geq S_2$.

são simplificados da sequinte forma:

(1) $s_1 = s_1$ & (S_2).

(2) $s_1 = s_1$ | (S_2).

Exemplo

O axioma:

datum $\geq$ integer.

ao ser simplificador se transforma em:

datum = datum **|** integer.

□

Regra 2.4

Axiomas da forma

(1) $s_1 = S_2$ **|** ... **|** $(S_n$ & ... & S_{n+m}).

(2) $s_1 = S_2$ & ... & $(S_n$ **|** ... **|** S_{n+m}).

são simplificados da seguinte forma:

(1) $r = S_n$ & ... & S_{n+m}.

$s_1 = S_2$ **|** ... **|** S_{n-1} **|** r.

ou

(2) $r = S_n$ **|** ... **|** S_{n+m}.

$s_1 = S_2$ & ... & S_{n-1} & r.

Exemplo

O axioma:

natural = 0 **|** (positive-rational & integer).

é traduzido para a notação simplificada como:

natural = 0 **|** r.

r = (positive-rational & integer).

□

Regra 2.5

Axiomas da forma

$e : S_1.$

são simplificados da seguinte forma:

$r = S_1.$

$e : r.$

Exemplo

O axioma:

0 : (natural $\&$ boolean).

é traduzido para a notação simplificada como:

0 : novo-sorte.

novo-sorte = (natural $\&$ boolean).

$\square$

3.4.1 Exemplo de Especificação Simplificada

A conversão do exemplo das pilhas definido na Seção 2.3, é feita através dos seguintes passos:

- A especificação dos naturais é adicionada à especificação simplificada pois é necessária para a especificação de pilhas.
- O seguinte axioma:

  item $\geq$ natural.

não é permitido em uma especificação algébrica unificada simplificada. Por isso ele deve ser reescrito para o seguinte axioma válido:

item = (item | natural).

Com isso obtemos a seguinte especificação:

introduces: stack , empty-stack , item , natural , nothing , 0 , push _ _ , pop _ , top _ , sucessor _ , sum _ _ .

(1) push _ _ :: item , stack → stack.

(2) pop _ :: stack → stack.

(3) top _ :: stack → item.

(4) sucessor _ :: natural → natural.

(5) sum _ _ :: natural , natural → natural (*commutative*, *unit is* 0).

(6) fib _ :: natural → natural.

(7) item = (item | natural)

(8) top push x y = x

(9) pop push x y = y

(10) top empty-stack = nothing

(11) pop empty-stack = empty-stack

(12) 0 : natural

(13) sum 0 n = n

(14) sum sucessor m n = sucessor sum m n

(15) fib 0 = 0

(16) fib sucessor 0 = sucessor 0

(17) fib sucessor sucessor x = sum (fib x) (fib sucessor x)

□

3.5 Tradução de Especificações Algébricas Unificadas Simplificadas

Como foi dito no início desse capítulo, a tradução de especificações algébricas[1] se baseará nas seguintes similaridades entre os conceitos existentes nas especificações algébricas e no paradigma orientado a objetos:

1. Os conceitos de objetos e classes são responsáveis pela representação de sortes
2. Os conceitos de herança e subclasses representam a relação de inclusão de sortes.
3. Os métodos representam operadores definidos por especificações algébricas.

A tradução de uma especificação algébrica em código orientado a objeto se dá através das seguintes etapas:

1. Identificação dos sortes.
2. Geração de um esqueleto de classes.
3. Geração dos métodos auxiliares.
4. Geração dos métodos operadores.

Essas etapas serão detalhadas nas próximas seções. Durante a explicação de cada etapa utilizaremos como exemplo segmentos da especificação de pilhas de naturais, cuja simplificação é mostrada na Seção 3.4.1. A tradução detalhada e comentada de toda a especificação é apresentada no Apêndice A.

[1] Durante essa Seção usaremos o termo especificações algébricas para se referir a especificações algébricas unificadas simplificadas.

Etapa 1: Identificação dos Sortes

Essa etapa tem como objetivo principal a identificação dos sortes definidos pela especificação corrente e quais as relações de igualdade e inclusão existentes entre esses sortes. Essas informações serão úteis para a geração do código orientado a objetos nas próximas etapas.

O primeiro passo para a identificação dos sortes existentes consiste em definir as constantes existentes na especificação fonte. As constantes definidas pela especificação são definidas pelos:

1. Termos válidos constantes existentes na especificação;
2. Declarações de funcionalidades. Uma declaração de funcionalidade da forma:

   (1) $o :: s_1, s_2, \ldots \rightarrow s_n.$

   define a constante: $o(s_1, s_2, \ldots)$, que é subsorte do sorte s_n;
3. As constantes que não estiverem na forma normal devem ser reescritas até satisfazerem essas condições.

O segundo passo do processo de identificação consiste em agrupar as expressões obtidas no passo anterior que representarem os mesmos sortes. Isso é feito através de:

1. Análise dos axiomas de igualdade da especificação.
2. Uso dos axiomas da igualdade:

   $x = x.$

   $x = y \;\Rightarrow\; y = x.$

   $x = y \wedge y = z \;\Rightarrow\; x = z.$

   $x \leq y \wedge x \geq y \;\Rightarrow\; x = y.$

O terceiro passo consiste em definir a relação de inclusão. Isso é feito através das seguintes regras:

1. O sorte nothing é subsorte de todos os sortes.

2. Todo sorte é subsorte e supersorte dele mesmo.

3. Se s é um sorte gerado pela expressão "$s_1 \mid s_2$", então s é um supersorte de s_1 e s_2.

4. Se s é um sorte gerado pela expressão "$s_1 \mid s_2$", então todo supersorte de s_1 e s_2 simultaneamente é também um supersorte de s.

5. Se s é um sorte gerado pela expressão "s_1 & s_2", então s é um subsorte de s_1 e s_2.

6. Se s é um sorte gerado pela expressão "s_1 & s_2", então todo subsorte de s_1 e s_2 simultaneamente é também um subsorte de s.

7. Uso da propriedade de monotonicidade dos operadores:

   $$x \leq y \;\Rightarrow\; \mathsf{o}(x) \leq \mathsf{o}(y).$$

8. Uso dos axiomas da relação de inclusão de sorte.

   $$x \leq y \wedge y \leq z \;\Rightarrow\; x \leq z.$$
   $$x \leq y \;\Rightarrow\; y \geq x.$$

Finalmente, o quarto passo define quais sortes da especificação são indivíduos. Para isso utilizamos as seguintes regras:

- É analisada, na especificação fonte, os axiomas da forma:

  (1) $x : y.$

- Os sortes formados pela aplicação de operadores de atributos *total* e com indivíduos como argumentos também são indivíduos.

A aplicação desses passos produzirá informações sobres os sortes definidos pela especificação e quais as suas principais características. Essas informações serão utilizadas na geração do código orientado a objetos.

Exemplo

A especificação simplificada das pilhas define, entre outros, os seguintes sortes e relações:

1. Sorte 0: subsorte dos sortes natural e item.
2. Sorte natural: subsorte do sorte item.
3. Sorte stack: que não possui supersortes.
4. Sorte gerado pelo termo push item stack, subsorte de stack, que representa as pilhas não vazias.
5. Sorte empty-stack: indivíduo do sorte stack.

□

Etapa 2: Geração do Esqueleto de Classes

A partir dessa etapa, começamos a construção do código que traduz a especificação fonte. O processo de geração do esqueleto de classes consiste na definição de:

- Os objetos globais que representarão os sortes da especificação.
- Classes que serão geradas pelo processo de tradução.
- Relação de herança entre essas classes.

Os objetos produzidos pela especificação servirão para representar os sortes da especificação durante a representação de termos definidos nesta na forma de expressões de linguagens orientadas a objetos. Por esse motivo, todos os sortes definidos pela especificação possuirão, no código produzido pelo processo de tradução, um objeto que o representará.

As classes geradas podem ser de quatro tipos distintos:

- Classes genéricas: uma classe genérica gerada por um sorte s é responsável por modelar as características comuns a todos os subsortes desse sorte, inclusive ele próprio.

- Classes específicas: uma classe específica gerada por um sorte s é responsável as características do sorte s que não são compartilhadas por seus subsortes. Nos casos em que o sorte s não possuir subsortes próprios, a funcionalidade das classes genéricas e específicas podem ser agrupadas em uma única classe.

- Classes operadoras: classes operadores são geradas pelos operadores da especificação para representar termos não-reduziveis, produzidos através do uso desses operadores.

- Classes especiais: geradas automaticamente pelo processo de tradução de forma independente da especificação. Como exemplos de classes especiais podemos citar as classes `Sort` e `Join` (ver explicação adiante).

A geração do esqueleto de classes é feita a partir das informações de sortes obtidas pela Etapa 1 utilizando as seguintes regras:

1. As seguintes classes especiais serão geradas:

   (a) Classe `Sort`: superclasse de todas as classes produzidas pelo processo de tradução.

   (b) Classe `Join`: subclasse da classe `Sort`, que representará os sortes formados pela união de dois ou mais subsortes e cujo resultado não corresponde a nenhum sorte definido pela especificação.

2. O processo de tradução produzirá, para todos os sortes definidos pela especificação que não puderem ser reescritos, gerarão uma classe específica e uma classe genérica.

3. A classe específica produzida pela tradução de um sorte s é subclasse da classe genérica produzida pela tradução do mesmo.

4. A classe genérica gerada a partir de um sorte s é subclasse das classes genéricas geradas pelos supersortes de s.

5. A tradução dos sortes gerados pela aplicação de um operador o com argumentos $s_1, s_2, ..., s_n$, que não forem indivíduos, produzirá uma classe operadora para representar os termos produzidos pela aplicação do operador o a subsortes dos sortes $s_1, s_2, ...$. Essa classe possuirá as seguintes características:

    - A classe operadora, gerada pela tradução de um sorte s, é subclasse das classes genéricas e classes operaradoras produzidas pela tradução dos supersortes de de s.
    - As classes operadoras geradas definirão variáveis de instância, necessárias para armazenar o valor dos argumentos do termo representado, e um método de nome `Init` que inicializa essas variáveis. Essas variáveis e método podem estar definidas no corpo da classes ou serem herdados de suas superclasses.

6. A classe específica do sorte nothing, se formos seguir as regras definidas acima, seria subclasse de todas as classes existentes o que pode gerar problemas pois a maioria dos compiladores existentes no mercado costumam limitar o número de superclasses imediatas que uma classe pode conter. Como forma de amenizar esse problema, especificaremos que essa classe deve ser subclasse apenas dos sortes genéricos que forem parâmetros de operadores. A necessidade dessa relação se deve ao fator que o sorte nothing pode ser passado como argumento de um operador no lugar de qualquer sorte. Uma outra abordagem possível seria a tradução do sorte nothing como sendo o valor `NULL` de C++, entretanto essa abordagem possui com inconveniete principal o fator de que a chamada de métodos para o objeto `NULL` frequentemente produz exceções que abortam a execução do programa.

Durante o resto desta tese utilizaremos as seguintes convenções sintáticas:

- `G(`s`)`: designa a classe genérica gerada pelo sorte s.
- `E(`s`)`: designa a classe específica do sorte s.
- `Sort_`s: denota uma variável global que traduz o sorte s.

Exemplo

A especificação de pilhas de inteiros, mostrada na Seção 3.4.1, produz um esqueleto de classes cuja hierarquia completa é mostrada na Figura 3.3.

Os sortes: (natural, stack e item, produzirão classes genéricas e específicas que representarão as propriedades destes.

As classes geradas a partir de operadores, como `(sucessor natural)`, representam os sortes resultantes do operador sucessor que não puderam ser reduzidos segundo os axiomas e contém variáveis de classe para representar seus parâmetros. Como podemos observar a tradução definirá algumas classes que não serão utilizadas como por exemplo as classes `(pop stack)`, `(sum natural natural)`. Essas classes não produzirão nenhum objeto, pois os axiomas são capazes de reduzir todas as expressões da forma pop x e sum x y em objetos de outras classes, o que não ocorre com expressões da forma sucessor x por exemplo. A geração dessas classes de deve ao fato de que não existência de um algoritmo que decida se uma classe é realmente utilizada.

Além dessas classes, serão gerados os seguintes objetos:

- `Sort_stack`, objeto da classe `E`(stack).
- `Sort_natural`, objeto da classe `E`(natural).
- `Sort_item`, objeto da classe `E`(item).
- `Sort_empty_stack`, objeto da classe `E`(empty-stack).
- `Sort_0`, objeto da classe `G(0)`.

□

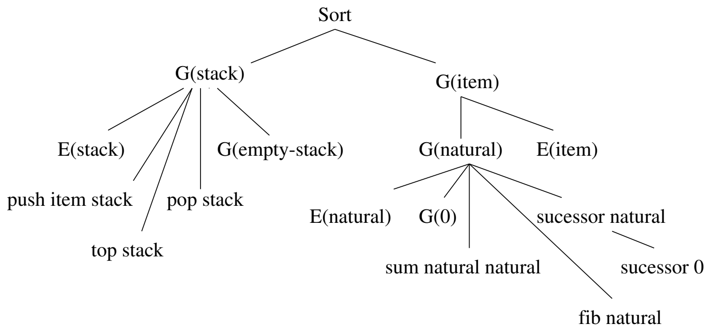

Figura 3.3: Hierarquia Simplificada de Classes Gerada pela Especificação de Pilhas

Etapa 3: Geração dos Métodos Auxiliares

Os métodos auxiliares implementam o comportamento comum a todos os sortes independentemente de especificações. A função desses métodos é responder a perguntas sobre a relação de inclusão de sortes e auxiliar a execução dos métodos operadores.

Todas as classes definidas na etapa anterior responderão aos seguintes métodos auxiliares:

- `bool Match_`*name*`(Sort & p1, Sort & p2,...)`: Estes métodos retornam verdadeiro se o objeto que receber a mensagem representar um termo produzido pela aplicação do operador *name*. Caso seja verdade, o método devolverá o valor os respectivos argumentos nas variáveis `p1, p2, ....`

- *name* `The`*name*`()`: Esses métodos retornam o objeto que recebe a mensagem, se o sorte representado por este objeto for um subsorte do sorte *name*, caso contrário ele retorna o valor `NULL`.

- `bool Is(Sort s)`: Esse método retorna o valor `true` se o objeto corrente for igual ao parâmetro fornecido.

- `bool IsSubSorte(Sort s)`: Esse método retorna o valor `true` se o parâmetro for um subsorte do objeto corrente.

- `bool IsSuperSorte(Sort s)`: Esse método retorna o valor `true` se o parâmetro for um supersorte do objeto corrente.
- `bool IsIndividual()`: Retorna o valor `true` se o objeto representar um indivíduo.
- `bool IsIndividualOf(Sort s)`: Retorna o valor `true` se o objeto representar um sorte indivíduo e for um subsorte de `s`.
- `int OpComp(param)`: Esse é um método auxiliar usado pelos métodos `Is`, `IsSubSorte` e `IsSuperSorte`, e retorna os seguintes valores:
  - -1 : Se o sorte do objeto corrente representar um subsorte do sorte representado por $name$.
  - 0 : Se o sorte do objeto corrente for exatamente o indicado por $name$.
  - 1 : Se o sorte do objeto correntes for um subsorte do sorte $name$.
  - Qualquer outro valor, se não existir nenhuma relação de inclusão entre os sortes.

  Para auxiliar a realização dessas comparações as seguintes estruturas foram definidas:
  - Um método `GetID`, que retorna um identificador que indica o tipo de sorte que o objeto representa.
  - É criado um objeto comparador que armazena informações sobre inclusão de sortes que serão consultadas pelo programa.
- `Sort Meet(Sort s)`: Esse método retorna um sorte formado pela interseção do sorte corrente com o sorte passado como parâmetro.
- `Sort Join(Sort s)`: Esse método retorna um sorte formado pela união do sorte corrente com o parâmetro fornecido

A geração dos métodos auxiliares varia de acordo com o tipo da classe segundo as seguintes regras:

Regra 4.1: Métodos Auxiliares da Classe `Sort`

A classe `Sort` possuirá o seguinte conjunto de métodos pré-definidos:

- `Match_`*name*`(Sort & p1, Sort & p2, ...)` : A classe `Sort` possuirá uma versão de `Match` para toda operação existente na especificação e seu comportamento será retornar o valor `false`.
- `The`*name*`()` : A classe `Sort` possuirá uma versão dos métodos `The` para todo o sorte existente na especificação e o seu comportamento será retornar o valor `NULL`.
- `Is(Sorte s); IsSubSorte(Sorte s); IsSuperSorte(Sorte s)` : Utilizam o resultado da operação `OpComp` para determinar o valor do resultado.
- `OpComp(Sorte s)`: Por default chamam o método `GetID` do objeto corrente e do argumento e passa os valores retornados para o objeto comparador (descrito mais adiante).
- `IsIndividual()` : Retorna o valor `false`.
- `IsIndividualOf(Sort s)` : Chama as funções `IsIndividual` e `IsSubSorte` para determinar o resultado.
- `Sort Meet(Sort s)`: O comportamento desse método é retornar o menor entre os sortes comparados `s` ou `this`.
- `Sort Join(Sort s)`: O comportamento desse método é retornar o maior entre os sortes comparados `s` ou `this`. O comportamento desse método, assim como o do método `meet` gera resultados imprecisos, entretanto a implementação correta desses métodos exigiriam um overhead no código gerado que não foi encontrada uma solução satisfatória.

Regra 4.2: Métodos Auxiliares das Classes Genéricas

Todo o sorte *name* da especificação define os seguintes métodos para sua classe genérica:

- `The`*name*`()`: Esse método será redefinido para retornar o valor `this`.

Regra 4.3: Métodos Auxiliares das Classes Específicas

Toda a constante da especificação de nome *name* define os seguintes métodos para sua classe específica:

- `Match_`*op*`(Sort & p1, Sort & p2,...);`: É gerado um método `Match_`*op* para cada operador *op* que gerar essa constante. Sua funcionalidade será a de atribuir os valores correspondentes para os argumentos e, a seguir, retornar o valor `true`.

- `IsIndividual()`: Se *name* for um indivíduo, esse método é redefinido para retornar o valor `true`.

- `int GetID()`: Retorna um identificador para esse sorte.

Regra 4.4: Métodos Auxiliares das Classes Operadoras

Toda classe operadora, gerada pelo operador *op*, possuirá os seguintes métodos:

- `Match_`*op*`(Sort & p1, Sort & p2,...)`: Esse método possui como funcionalidade alterar o valor das variáveis `p1, p2, ...` a retornar o valor `true`.

- `IsIndividual()`: Se *op* for um operador total, esse método deve ser reescrito de forma a verificar se os parâmetros do termo representado são também indivíduos antes de retornar uma resposta decisiva.

- `OpComp(Sort s)`: Esse método é redefinido de forma a analisar também o caso de inclusão de sortes por meio da monotonicidade dos operadores.

Etapa 4: Geração dos Métodos Operadores

Os métodos operadores são gerados pela tradução das funcionalidades e dos axiomas responsáveis por definir o comportamento dos operadores. O principal objetivo desses métodos será verificar se um termo gerado pela aplicação do operador a um conjunto de operandos pode ser de acordo com os axiomas da especificação, que deverão necessariamente estar na forma de equações de reescrita. Se o termo puder ser simplificado o método operador devolverá o resultado dessa simplificação. Caso contrário será retornada um representação para esse termo utilizando a classe operadora correspondente.

Todo operador op, de funcionalidades:

$op :: s_{1,1}, s_{2,1}, \ldots \rightarrow s_{n,1}.$

$op :: s_{1,2}, s_{2,2}, \ldots \rightarrow s_{n,2}.$

$op :: s_{1,m}, s_{2,m}, \ldots \rightarrow s_{n,m}.$

gerará um método operador. Esse método operador possuirá a seguinte assinatura:

C_n op(C_2 `param2,...,`$C_{(n-1)}$ `param(n-1));`

onde os nomes C_i o nome da classe genérica do sorte mais genérico do i-ésimo parâmetro das funcionalidades ou ainda a classe pré-definida `Sort`, caso não seja possível definir um sorte mais genérico. As principais características desse método são:

- Pertencerá a classe genérica gerada pela tradução do sorte s_1.
- Receberá como parâmetros objetos das classes genéricas geradas pela tradução dos sortes s_2, $s_3, \ldots, s_{n-1}$.
- Retornará um objeto da classe genérica gerada pela tradução do sorte s_n.

O corpo desse método é estruturado da seguinte forma:

{
  constantes(op)
  axiomas(op)
  classes(op)
}

onde:

- constantes(op): é uma função de tradução que produzirá o código responsável por verificar se o operador produz uma das constantes identificadas na especificação fonte.
- axiomas(op): é uma função de tradução que produzirá o código responsável por verificar se existem regras de reescrita que tratem o termo formado pelo operador op com os parâmetros `this,param2,....`
- classes(op): é uma função de tradução que produzirá o código responsável por criar um objeto da classe operadora correta, se nenhuma equação de reescrita puder ser aplicada.

A definição de cada uma dessas funções será vista detalhadamente nas próximas seções.

Função de Tradução classes

Durante a geração do esqueleto de classes (Etapa 2) são definidos conjunto de classes operadoras. Um classe operadora tem como objetivo representar termos produzidos pela aplicação de um operador, submetido a um conjunto limitado de parâmetros. A função de tradução classes produzirá o código responsável pela criação de objetos das classes operadoras.

O código produzido pela função de tradução classes(op) é definido através dos seguintes passos:

1. A função classes produzirá, para toda cada classe operadora `C` responsável por representar sortes produzidos pela aplicação do operador op a argumentos subsortes de s_1, s_2, etc, um código que verificará se os argumentos utilizados na chamada ao método estão de acordo com as restrições de sorte da classe operadora. Se o resultado do teste for positivo, então será criado uma nova instância da classe operadora `C` que será o devolvido como resultado da execução do método operador. O código gerado por esse processo possui o seguinte formato:

```
if (this->IsSuperSorte(Sort_s1))
  if (param2->IsSuperSorte(Sort_s2))
    .....
```

```
return new C(this,param2,...);
```

2. Os segmentos produzidos pelo passo anterior são concatenados, tendo cuidado de posicionar os testes mais específicos no início do código (vide exemplo). O segmento de código produzido definirá o resultado do operador op.

Exemplo

Como exemplo da função de tradução classes(op), o código gerado pela expressão:

classes(sucessor _)

assumindo que o operador sucessor _ possui as seguintes funcionalidades:

(1) sucessor _ :: integer $\rightarrow$ integer.

(2) sucessor _ :: natural $\rightarrow$ natural.

Produz os seguintes segmentos de código:

1.
```
if (this->IsSuperSorte(Sort_integer))
      return new C(this);
```

2.
```
if (this->IsSuperSorte(Sort_natural))
      return new D(this);
```

Nesses segmentos de código as classes C e D representam termos gerados pela aplicação do operador sucessor _ com parâmetros dos sortes integer e natural, respectivamente.

O código final gerado pelo operador classes é formado pela concatenação dos segmentos mostrados acima. O processo de concatenação desses segmentos deve ser feito de forma a respeitar a relação inclussão de sortes, evitando que a precedência dos testes mais genéricos. No exemplo acima a concatenação dos segmentos na ordem mostrada irá ocasionar erros de execução pois gera objetos da classe C para argumentos do sorte natural. Esse erro ocorre pois o sorte natural é subsorte do sorte integer. Para corrigir esses tipos de erros os segmentos são reorganizados produzindo o seguinte resultado:

```
if (this->IsSuperSorte(Sort_natural))
      return new C2(this);
if (this->IsSuperSorte(Sort_integer))
      return new C1(this);
```

□

Função de Tradução constantes

O segmento de código definido pela função de tradução constantes(op) é responsável por verificar se existem constantes geradas por aplicações do operador op. O código gerado pela função é definido a partir dos seguintes passos:

1. Verificar quais das constantes identificadas na Etapa 1 é produzida por uma operação op.
2. Assumindo que c é uma constante gerada pelo pela aplicação do operador op aos parâmetros x_1, x_2, etc, o seguinte código é gerado:

```
{
   if (this->Is(Sort_x1) && param2->Is(Sort_x2) ...  )
     return Sort_c;
}
```

3. O resultado do operador será a união de todos os segmentos produzidos pelo passo anterior.

Exemplo

Como exemplo seja o axioma:

pop empty-stack = nothing.

ao identificarmos as constantes da especificação, descobrimos que o sorte nothing é gerado pela expressão pop empty-stack. Por isso a função constantes(pop_), produzirá, entre outros o seguinte trecho de código:

```
{
   if (this->Is(Sort_empty-stack))
     return Sort_nothing;
}
```

□

Função de Tradução axiomas

O segmento de código gerado através da função de tradução axiomas(op) é responsável por verificar a possibilidade de realização de reescritas no termo representado.

O código gerado pela função axiomas(op) é formado pela concatenação dos segmentos de código produzidos pelos axiomas da forma:

(1) $rel \;\Rightarrow\; op(x_1,x_2,...,x_n) = e.$

onde:

- rel: é o conjunto de zero ou mais sentenças relacionais.
- e: é um termo da especificação.
- $x_1,...$: são termos da especificação.

o código produzido por cada axioma ter por função verificar se essa equação de reescrita pode ser aplicada ao termo que está sendo construído pela chamada ao método operador e, se possível, aplica a regra de reescrita. Para realizar essa tarefa, o código produzido:

1. Verifica se os argumentos recebidos pelo método operador são correspondentes aos indicados pelos termos x_1,x_2,...,x_n e se os argumentos satisfazem as condições especificadas em rel. Durante esses testes são definidas variáveis temporárias que armazenam o valor das termos variáveis existentes no axioma ou ainda resultados de cálculos intermediários.

2. Se os testes anteriores forem satisfeitos o termo e será produzido e retornado como resultado do método operador.

o esquema do código produzido por axiomas dessa forma é mostrado abaixo:

```
{
    l1;
    l2;
    ...
    ln;
    ln+1;
    if (t1 && t2 && ...  && tn && tn+1)
    {
        return av;
    }
}
```

Onde:

1. m_0 =empty-map

2. (l_1, t_1, m_1) = test-term(p_1, x_1, m_0)

3. (l_2, t_2, m_2) = test-term(p_2, x_2, m_1)

4. ...

5. (l_n, t_n, m_n) = test-term(p_n, x_n, m_{n-1})

6. (l_{n+1}, m_{n+1}) = test-rel(rel, m_n)

7. av = evalue(e, m_{n+1})

No código acima:

- Os símbolos l_x representam declarações de variáveis locais utilizada para armazenar resultados temporários.
- Os símbolos t_x representam expressões C++ encarregadas de realizar os testes especificados pelos axioma.
- Os símbolos m_x representam mapeamentos entre nomes de variáveis e expressões que armazenam o seus respectivos valores.
- Os operadores test-term, test-rel e evalue são funções auxiliares que descreveremos a seguir.

Função Auxiliar: test-term

A função auxiliar test-term recebe como argumentos a tupla: (s,e,m), onde:

- s representa um termo válido na especificação fornecida ao tradutor.
- e representa uma expressão de uma linguagem orientada a objetos.
- m representa um mapeamento entre nomes de variáveis e expressões orientadas a objetos, esse mapeamento tem como objetivo representar o valor correspondente a cada variável conhecida.

e produz como resultado a tupla (l,t,m'), onde:

- l: é um conjunto de declarações de variáveis locais.
- t: é um teste responsável por verificar se o sorte avaliado pela expressão e corresponde ao sorte especificado pelo termo s.
- m': é o mapeamento formado pelo argumento m acrescido de novas associações que possam ter sido criadas durante o processamento do operador.

cujos valores são definidos pelas seguintes regras:

- Se todas as variáveis existentes no termo s possuírem valores conhecidos, ou seja, todas as variáveis em s pertencem ao domínio de m, verificamos se a expressão e corresponde ao termo s através da utilização do método pré-definido `IS`, que comparará o objeto avaliado pela expressão e com o objeto avaliado pela expressão equivalente ao termo s. Nesses casos os valores de retorno são os seguintes:

  $l =$ empty.

  $m' = m$.

  $t =$ (e`->IS(`e'`))`

  $e' =$ Avaliar(s,m)

- Se s é formada por uma variável de valor desconhecido, ou seja, que não pertence ao domínio de m, então, a comparação é sempre verdadeira uma vez que a variável s ainda não foi definida. O efeito da função auxiliar, será o associar o valor da variável s à expressão retornada pela avaliação de e. O resultado produzido pela função será:

  $l =$ empty.

  $m' =$ overlay m (map s to e).

  $t =$ `true`.

- Se s é um termo da forma $op(p_1,p_2,...,p_n)$, a comparação deve:

  1. Verificar se o termo produzido pela expressão e é um termo produzido pelo operador op, através do método `Match_`op, que para ser executado necessitará da definição de um conjunto de variáveis locais para armazenar os argumentos do termo representado pela expressão e.
  2. Caso o primeiro teste seja satisfeito, é necessário ainda verificar se os argumentos p_1,...p_n, correspondem aos argumentos da expressão e. Essa comparação é definida através de chamadas recursivas à função de tradução test-term.

  Nesses casos o resultado produzido pela função de tradução será:

  $l =$
  `Sort` $local_1$`,`$local_2$`, . . . ,`$local_n$`;`$l_1 l_2 ... l_n$

  $t =$ e`->Match_`op`(`$local_1$`,`$local_2$`, . . . ,`$local_n$`) &&` t_1 `&&` t_2 `. . .` t_n

$(l_1,t_1,m_1) = \textsf{test-term}(p_1,local_1,m)$.
$(l_2,t_2,m_2) = \textsf{test-term}(p_2,local_2,m_1)$.
...
$(l_n,t_n,m') = \textsf{test-term}(p_n,local_n,m_{n-1})$

Função Auxiliar: test-rel

A função auxiliar test-rel recebe como argumento a tupla (rel,m), onde:

- rel é uma conjunção de zero ou mais equações válidas em especificação algébricas unificadas simplificadas.
- m é um mapeamento entre variáveis e expressões orientadas a objetos, responsável por armazenar o valor das variáveis conhecidas.

como resultado, a função auxiliar produz a tupla (l,t,m'), onde:

- l: é um conjunto de declarações de variáveis locais.
- t: é uma expressão capaz de verificar se o operador satisfaz as equações existentes no argumento rel.
- m': armazena o valor das variáveis conhecidas após a execução da função auxiliar.

Esses valores são definidos pelas seguintes regras:

- Se rel, não possuir nenhuma equação, então não existem testes a serem realizados. Portanto o resultado da função será:

  $l = \textsf{empty}$.
  $t = \texttt{true}$.
  $m' = m$.

- Se rel é formado pela conjunção de mais de uma equação, então cada uma destas devem ser testadas individualmente. Os valores de retorno podem ser obtidos através de chamadas recursivas à função test-rel, com isso os valores de retorno são definidos pelas seguintes expressões.

  $rel = rel_1 \wedge rel_2 \wedge ... \wedge rel_n$.

$l = l_1\ l_2\ ...\ l_n$.

$t = t_1$ `&&` t_2 `&&` ... `&&` t_n.

$(l_1,\ t_1,\ m_1) =$ test-rel(rel_1, m).

$(l_2,\ t_2,\ m_2) =$ test-rel(rel_2, m_1).

...

$(l_n,\ t_n,\ m') =$ test-rel(rel_n, m_{n-1}).

- Se rel é uma equação da forma:

(1) $exp = x$ "=" y.

o teste deve verificar se os termos x e y produzem termos iguais. Essa comparação pode ser feita utilizando a função auxiliar **test-term**. Para que esse teste possa ser traduzido corretamente, é necessário que pelo menos um dos termos não possua variáveis livres. Nesse caso os resultados produzidos pela função auxiliar são definidos pela seguinte expressão:

$(l, t, m) =$ test-term$(x,$evalue$(y,m),m)$.

- Se rel é uma equação da forma:

$exp = x$ "=" y_1 | y_2 | ... | y_n.

o teste deve verificar se o termo x é supersorte de um dos termos y_n, isso é feito através de chamadas ao método `IsSubSort`. Os resultados da função auxiliar **test-rel** para eses casos são definidos pelas expressões abaixo:

$l =$ Sort $local$;

$m' = m$.

$t =$ (($local$=evalue(x,m))`->IsSubSort`(evalue(y_1,m)) `&&` $local$`->IsSubSort`(evalue(y_2,m))
... $local$`->IsSubSort`(evalue(y_n,m)))

- Se rel é uma equação da forma:

$exp = x$ "=" y_1 & y_2 & ... & y_n.

o teste deve verificar se o termo x é subsorte de um dos termos y_n, isso é feito através de chamadas ao método `IsSuperSort`. Os resultados da função auxiliar **test-rel** para eses casos são definidos pelas expressões abaixo:

$l =$ Sort $local$;

$m' = m$.

$t =$ (($local$=evalue(x,m))`->IsSuperSort`(evalue(y_2,m)) `&&` $local$`->IsSuperSort`(evalue(y_2,
... $local$`->IsSuperSort`(evalue(y_n,m)))

Função Auxiliar: evalue

A função **evalue**, recebe como argumento a tupla (s,m), onde:

- s: é um termo válido pela especificação processada.
- m: é o mapeamento de nomes de variáveis para expressões orientadas a objetos, responsável por definir os valores das variáveis existentes no termo s.

O resultado produzido pela função é uma expressão orientada a objetos equivalente ao termo s. Essa expressão é obtida através das seguintes regras:

1. Se s é um nome de um sorte o resultado da função **evalue** será o objeto definido na etapa 2 para representar o sorte s.
2. Se s é uma variável o resultado será a expressão associada à variável s no mapeamento m.
3. Se s for um operador $op(p_1,...p_n)$, o resultado será a expressão:
   `Avaliar(`p_1`,`m`)->`op`(Avaliar(`p_2`,`m`),....)`

Exemplo

Como exemplo do processo de geração de método operador, iremos definir o método operador para o operador **fib _** da especificação exemplo. Esse operador possui as seguintes definições:

(1) fib _ :: natural $\rightarrow$ natural.

(2) fib 0 $=$ 0.

(3) fib sucessor 0 $=$ sucessor 0.

(4) $x =$ sucessor y $\Rightarrow$ fib sucessor $x =$ sum (fib x) (fib y).

os passos anteriores do processo de tradução definem que:

1. Os termos 0, sucessor 0, fib 0 e fib sucessor 0 são constantes e possuem variáveis associadas que denominaremos `Sort_0, Sort_s0, Sort_f0, Sort_fs0`, respectivamente.

2. A funcionalidade da linha 0 irá produzir uma classe operadora que denominamos `C_Fib`.

baseado na funcionalidade mais genérica do operador (definida pela funcionalidade da linha 0), é definido que o método operador para fib (que denominaremos `Fib`) pertencerá a classe `natural` (classe genérica do sorte natural) e possuirá o seguinte cabeçalho:

```
natural Fib()
```

o corpo do método é formado pelos código produzidos pelas seguintes funções tradução:

1. A função de tradução classes, analisará a constante produzida pela equação 1 e gerará o sequinte segmento de código:

```
if (this->IsSubSorte(Sort_natural))
   return new C_Fib(this);
```

2. A função de tradução constantes, analisará as equações constantes 1 e 2, produzindo o seguinte código:

```
if (this->IS(Sort_0))
   return Sort_0;
if (this->IS(Sort_s0))
   return Sort_s0;
```

3. A função de tradução axiomas, analisará a equação 3, e produzirá um código que verificará se a equação de reescrita pode ser aplicada. Para isso:

   (a) É gerado um teste para verificar se o termo se adequa ao lado esquerdo da igualdade (fib sucessor x =), o teste resultante é a seguinte expressão orientada a objetos: `this->Match_sucessor(local1)`, onde `local1` é uma nova variável local. Outro efeito desse passo é a associação da variável x a variável `local1` para usos futuros.

   (b) É gerado um teste para verificar a implicação existente na equação (x = sucessor y $\rightarrow$), gerando o seguinte teste: `local1->Match_sucessor(local2)`, onde:

a referência a variável `local1` se deve ao fato da variável x estar associada a esta. Durante esse passo é gerada uma nova variável de nome `local2` que estará associada à variável y.

(c) É gerada uma tradução para o termo do lado direito da igualdade, usando-se as associação de variáveis produzidas pelos passos anteriores. O resultado desse passo é a expressão: `local1->Fib()->Sum(local2->Fib())`.

(d) Finalmente as expressões produzidas pelos passos anteriores são unidas produzindo o seguintes segmento de código:

```
Sort local1,local2;
if (this->Match_sucessor(local1) && local1->Match_sucessor(local2))
   return local1->Fib()->Sum(local2->Fib());
```

como a equação 3 é a única equação do problema, o resultado dos passos anteriores será o código produzido pela função de tradução axiomas.

□

3.6 Otimizações no Código Gerado

Essa seção tem como objetivo propor algumas otimizações no código produzido pelo processo de tradução OO de especificações algébricas. Essas otimizações foram definidas através da análise das características do código produzido pela tradução de especificações.

Otimização 1: Eliminação de Chamadas `Is`

Chamadas ao método `Is` da forma:

x`->Is(Sort_`y`)`

podem ser substituídas por:

- x`==Sort_`y

Otimização 2: Eliminação de Chamadas `IsSubSorte`

Chamadas ao método `IsSubSorte` da forma:

x`->IsSubSorte(Sort_`y`)`

podem ser substituídas por:

- Constante `true`, se o tipo da expressão x for uma subclasse da classe genérica do sorte x.
- Expressão x`->The_`x`()!=NULL`, caso contrário.

Otimização 3: Utilização de Polimorfismo

Trechos de código como:

`if (this==Sort_`x `&&` *teste*`)`
    comando

onde:

- *teste*: é uma expressão de comparação, e
- *comando*: é um comando genérico

são comuns nos métodos gerados pelos operadores.

Comparações da forma `this==Sort_`x, só serão verdadeiras para objetos da classe especifica gerada pelo sorte x que é a responsável pela implementação do objeto `Sort_`x. Como forma de melhorar a execução dos métodos operadores podemos redefinir o método operador na classe específica gerada pelo sorte x. A nova versão do método será responsável pela realização de testes como os descritos acima, reduzindo o número de comparações do corpo principal do método operador.

Otimização 4: Eliminação de Testes Repetidos

O seguinte código:

```
if ( teste1 && teste2)
{
   comando1
}
if ( teste1 && teste3)
{
   comando2
}
```

é caracterizado pelo fato da expressão $teste_1$ ser calculada várias vezes durante a execução do programa. Um característica do código gerado pelo processo de tradução é a não geração de instruções que alterem o estado de um objeto ou variável, depois que estes são inicializados. Dessa forma, o valor retornado pelas avaliações da expressão $teste_1$ são idênticos. Por esse motivo, segmentos de código como os mostrados acima podem ser reescritos nas seguintes formas otimizadas:

```
if ( teste1 )
{
   if ( teste2)
   {
     comando1
   }
   if ( teste3)
   {
     comando2
   }
}
```

3.7 Limitações do Método

O processo de tradução descrito nesse capítulo possui algumas restrições que não o tornam capaz de fornecer a tradução de todas as especificações algébricas unificadas válidas. Algumas dessas restrições foram colocadas devido à impossibilidade de se implementar alguns tipos de axiomas válidos, ou ainda quando a implementação desses axiomas produziria um aumento de complexidade do código produzido que inviabilizaria a sua utilização.

As próximas seções mostrarão os tipos de declarações que não serão tratadas pelo tradutor.

3.7.1 Assinatura

O processo de tradução possuirá as seguintes restrições sobre as assinaturas que poderão ser tratadas pelo método:

- Toda declaração de funcionalidade deve declarar como sortes dos argumentos e do resultado da operação apenas termos constantes. O modelo de especificação algébrico unificado não faz nenhuma ressalva quando a esse ponto, permitindo sentenças como:

  if _ then _ else _ :: truth-value, x, $y \rightarrow (x \mid y)$.

- Os operadores associativos não são tratados pelo método. Essa limitação foi provocada porque a sua utilização, principalmente combinada com o atributo de comutatividade, tornariam complexos os métodos resultantes da tradução de axiomas. Essa dificuldade de tradução pode ser exemplificada pelo axioma:

  sum 0 $x = x$.

  considerando a operação sum associativa e comutativa, o código produzido teria que lidar com expressões da forma: sum 0 x, sum x 0, sum (sum x 0) y, etc.

3.7.2 Axiomas

Os axiomas em uma especificação unificada têm duas finalidades principais:

- especificar a semântica dos operadores; e

- indicar as relações existentes entre os sortes.

Um axioma usado para indicar a semântica dos operadores possui um dos seguintes formatos abaixo:

(1) $e_1(X) = e_2(X)$.

(2) $EQ_1 \Rightarrow EQ_2 \ldots \Rightarrow e_1(X) = e_2(X)$.

onde:

- $e_n(X)$ indica termos da especificação onde X é um conjunto de variáveis.
- EQ_n indicam sentenças válidas para a especificação.

o processo de tradução desses axiomas possuirá as seguintes limitações:

- Os axiomas devem estar na forma de um sistema de reescrita de termos terminante e confluente, como descrito no Capítulo 2, caso contrário resultados errados podem ser obtidos. O que torna axiomas como:

  $x = y * z \Rightarrow x / y = z.$

  não tratáveis pelo tradutor.

- Nos axiomas que se encaixarem no segundo formato, somente consideremos os casos em que se as condições EQ_1, EQ_2,..., EQ_n forem verdadeiras, então a igualdade $e_1(X) = e_2(X)$ deve ser verdadeira. Apesar do modelo permitir, não será considerada a possibilidade da igualdade $e_1(X) = e_2(X)$ ser falsa obrigar pelo menos uma das condições EQ_1, EQ_2,..., EQ_n também serem falsas, tornando as axiomas como:

  (1) $a{=}b \Rightarrow$ false.

  não traduzíveis.

Os axiomas que não indicarem a semântica dos operadores são utilizados para definir a relação de inclusão dos sortes. Nesse caso, as seguintes restrições ocorrerão:

- Essas equações não devem ser condicionais, essa limitação foi introduzida como forma de simplificar as relações existentes entre os sortes e consequentemente o código gerado. Essa característica impossibilita a tradução de axiomas da forma:

  (1) $x \ \mathsf{mod}\ y = \mathsf{0} \ \Rightarrow \ x \leq \mathsf{multiplos}\ y.$

- Pelo mesmo motivo, não serão permitidos axiomas como:

  (1) $\mathsf{sort} \geq x.$

  em que existam variáveis que não possuam nenhuma indicação de limites.

3.8 Implementação do Método

Como forma de avaliar melhor o método de tradução orientada a objetos foi feita uma implementação em C++ das etapas do processo descrito nesse capítulo. O programa tradutor, que implementa o método proposto, realiza as seguintes tarefas:

- Reconhece a sintaxe de uma especificação algébrica unificada geral.
- Executa as etapas de tradução descritas nesse capítulo, produzindo uma biblioteca orientada a objetos.
- Insere no código da biblioteca rotinas responsáveis pela desalocação automática de memória.
- Produz uma interface com o usuário (shell) para ser utilizada no teste da especificação fonte.

Além disso, como forma de melhorar o desempenho do método e fornecer uma melhor integração com trechos de código C++, foram inseridas na implementação as seguintes características:

- A capacidade de se definir classes e novos métodos adicionais aos produzidos pelo processo de tradução.
- A capacidade de definir sortes internos que serão tratados a parte pelo processo de tradução. Um conjunto de sortes internos é definido pelas seguintes propriedades:
  1. Uma expressão léxica que identifica esses sortes.
  2. O seu supersorte imediato.
  3. Uma regra de geração de código para sortes pertencentes a esse grupo.

essas características foram utilizadas na definição dos valores numéricos e das strings da notação de dados (Apêndice C). Como exemplo, os valores numéricos foram definidos da seguinte forma:

- Foi acrescentado uma classe de nome `InternalNumber`, subclasse da classe gerada pelo sorte integer, que armazena um inteiro representando um número, e possui métodos como: `sum_`, `difference_`, etc. Para tratar as operações sobre inteiros.
- Foram adicionadas ao tradutor a seguintes definições de um sorte interno:
  - Expressão léxica: Uma sequência de números.
  - Supersorte imediato: integer
  - Regra de geração para um sorte n: `new InternalNumber(`n`)`.

Exemplo de Uso do Programa Tradutor

Como exemplo de uso do tradutor de especificações algébricas unificadas, mostramos aqui a tradução da especificação de pilhas de naturais (Apêndice A) pelo programa tradutor, e a subsequente utilização do programa gerado são mostrados abaixo:

```
# tradutor stack
Analisando introduces do arquivo stack.as
Analisando needs do arquivo stack.as
Analisando introduces do arquivo natural.as
Lendo arquivo stack.as...
Lendo arquivo natural.as...
Especificacao lida !!
Traduzindo especificacao fonte...
Traducao Concluida !!
# gcc -I. -I.. stack.cc ../tools.cc
# ls -l
-rwxr-xr-x   1 lcsm     msc        192352 Jan 21 21:00 a.out*
-rw-r--r--   1 lcsm     msc           400 Oct  9 13:18 natural.as
-rw-r--r--   1 lcsm     msc           370 Oct  9 13:18 stack.as
-rw-r--r--   1 lcsm     msc         85791 Jan 21 20:59 stack.cc
-rw-r--r--   1 lcsm     msc         21169 Jan 21 20:59 stack.h
# a.out
Digite uma expressao:
> 0 <= natural
Resultado: true
Digite outra expressao:
> 0 <= stack
Resultado: false
Digite outra expressao:
> sum sucessor 0 sucessor 0
Resultado: sucessor(sucessor(0))
tempo gasto = 0.000000
Digite outra expressao:
> push a empty-stack
Resultado: push(a)(empty-stack)
tempo gasto = 0.000000
Digite outra expressao:
> p1 := push sucessor 0 empty-stack
Resultado: p1 = push(sucessor(0))(empty-stack)
tempo gasto = 0.000000
Digite outra expressao:
```

```
> p1 := push 0 p1
Resultado: p1 = push(0)(push(sucessor(0))(empty-stack))
tempo gasto = 0.010000
Digite outra expressao:
> top p1
Resultado: 0
tempo gasto = 0.010000
Digite outra expressao:
> pop p1
Resultado: push(sucessor(0))(empty-stack)
tempo gasto = 0.000000
Digite Outra Expressao:
> stop
#
```

O programa `tradutor` recebe como parâmetro o nome do arquivo que contêm a especificação a ser traduzida. A sua primeira tarefa é examinar quais outras especificações são requisitadas pela especificação e quais sortes e operadores estão definidos nestas, isso é feito através da análise das sentenças **needs** e **introduces** das especificações. Após essa identificação o programa carregará e unificará as especificações necessárias e traduzirá essas especificações, segundo as regras mostradas nesse capítulo. Após a tradução, é produzido um programa fonte em C++ que implementa a especificação. Esse programa pode então ser compilado por um compilador C++. O programa executável gerado possuirá um shell rudimentar que permite aos usuários realizarem perguntas sobre os sortes definidos pela especificação.

3.9 Comparações

Como teste da eficiência do método aqui proposto, utilizamos a função de cálculo dos números da série de Fibonacci de índices 10, 14 e 20, segundo a especificação mostrada na Seção 2.3. Como comparação utilizaremos um interpretador para OBJ3 [GKK+88] e um compilador de especificações em OBJ, descrito em [Ham95]. Os testes foram feitos utilizando-se uma Sun

SparcStation20 com 96M de memória principal, rodando o sistema operacional Solaris. Os resultados obtidos são mostrados na Tabela 3.1, em alguns casos (marcados com o símbolo ?) não foi possível obtermos os resultados devidos a estouros de pilhas. Podemos ver que a execução do programa escrito utilizando o método OO conseguiu uma melhoria de cerca de 11 vezes em relação ao sistema interpretado e 4 vezes em relação ao sistema compilado. Esse resultado demonstra a importância do método proposto nesse capítulo.

As principais vantangens do método, que ocasionaram esses resultados de performance, são listadas a seguir:

1. As regras de reescrita estão agrupadas em métodos que processam regras que simplificam um mesmo operador. Esse fator diminui o número de testes de regras. Por exemplo: O termo fib x, traduzido pela expressão x`->fib()`, ao ser executado, não testará a possibilidade de aplicação da regra:

   sum 0 $x = x$.

2. Somente os termos normais são construídos durante a execução do código, os termos não normais são representados de forma indireta pela pilha do sistema operacional.

Os tempos necessários para compilação e os tamanhos do código executável final, obtidos pelos dois métodos de compilação de especificações, são mostrados na Tabela 3.2. Esses resultados entretanto não dão uma idéia precisa da comparação entre os métodos devido ao tamanho reduzido da especificação fonte utilizada. Em especificações mais complexas, o número de classes no código orientado a objetos tende a se tornar elevado o que elevaria o tempo de compilação e tamanho do código objeto.

	fib(10)	**fib(14)**	**fib(20)**
OBJ interpretado	0,250	1,300	?
OBJ compilado	0,090	0,810	16.37
Tradução OO	0,020	0,180	3,86

Tabela 3.1: Comparação de Tempos de Execução (em segundos)

	Tam. Executável	**Tempo de Compilação**
OBJ compilado	171964	37
Tradução OO	102039	48

Tabela 3.2: Tamanhos de Executável (em bytes) e Tempos de Compilação (em segundos)

Capítulo 4

Semântica de Ações

Nesse capítulo descreveremos semântica de ações [Mos92, Wat91], formalismo utilizado para descrever a semântica de linguagens de programação. Inicialmente, como motivação para semântica de ações, faremos uma introdução geral sobre semântica de linguagens de programação. A seguir descreveremos as entidades definidas por semântica de ações e a forma proposta pelo modelo para a descrição da semântica de linguagens de programação utilizado. Finalizando, como exemplo ilustrativo, forneceremos a descrição de uma pequena linguagem imperativa utilizando semântica de ações.

4.1 Introdução

A descrição semântica de linguagens de programação através de textos escritos informalmente tende a gerar vários erros de interpretação devido à imprecisão da linguagem natural. Esses erros de interpretação podem fazer que diferentes compiladores/interpretadores, cuja construção foi baseada numa mesma especificação, gerem diferentes execuções para um mesmo programa.

Outra desvantagem do uso de descrições feitas em linguagem natural é a inexistência de métodos para prototipação segura dessas especificações, contribuindo para a geração de erros durante a fase de implementação.

O uso de métodos formais para a descrição da semântica de linguagens de programação seria uma arma poderosa para a redução desses problemas. Por isso vários métodos formais têm sido propostos com o objetivo de descrever a semântica de linguagens de programação. Entre os métodos propostos podemos citar: semântica denotacional [Sch86], semântica algébrica [Gue81], entre outros.

A maioria dos métodos propostos na literatura utilizam uma notação matemática de difícil compreensão ou ainda são baseados em linguagens dependentes de plataforma. Essas dificuldades tendem a inviabilizar o seu uso para a descrição de linguagens complexas como (ML, C++, Java, etc).

Semântica de ações [Mos92, Wat91] é um formalismo para descrição semântica de linguagens de programação derivado de semântica denotacional.

Semântica de ações difere de semântica denotacional por oferecer ao projetista um conjunto de entidades semânticas pré-definidas que modelam as principais estruturas de controle e de dados existentes em linguagens de programação. Essas entidades podem ser combinadas para expressar a semântica de linguagens de programação, evitando que o projetista tenha de especificar os conceitos básicos existentes em linguagens de programação como memória, abstrações, etc.

Outra característica de semântica de ações é o fator das entidades semânticas pré-definidas serem expressas utilizando uma notação bastante intuitiva, baseada em termos da língua inglesa. Por isso especificações em semântica de ações podem ser compreendidas por pessoas que não dominam o formalismo.

A notação formal utilizada por semântica de ações para a descrição de linguagens de programação é o modelo de álgebras unificadas, descrito no Capítulo 2. Dessa forma, a definição das entidades semânticas pré-definidas e a descrição de uma linguagem de programação são especificações algébricas unificadas. Essa característica permite a expansão do modelo através da adição de novas estruturas de dados e de controle.

Devido as suas características pragmáticas, semântica de ações tem sido utilizada para descrição de várias linguagens de programação reais. Entre alguns exemplos de especificações reais podemos citar descrições para Pascal [MW93] e SML [Wat88].

Semântica de ações também tem se mostrado útil para a geração automática de compiladores, capaz de gerar compiladores comparáveis aos escrito manualmente como veremos no Capítulo 5.

4.2 Notação de Ações

Semântica de ações disponibiliza ao projetista três tipos de entidades semânticas pré-definidas: dados, ações e produtores. Uma descrição informal de cada uma dessas entidades é mostrada nas próximas seções. Uma descrição mais formal da notação de ações, seus sortes e funcionalidades dos operadores pode ser encontrada nos Apêndices B e C.

4.2.1 Dados

A notação de dados define vários tipos de dados padrões que auxiliarão o especificador na descrição da semântica de um programa. A seguir citaremos os principais sortes e operadores definidos pela notação de dados conforme apresentador no Capítulo 5 de [Mos92].

Números

O modelo de ações define os seguintes sortes numéricos:

- natural: Conjunto dos números naturais.
- integer: Conjunto dos números inteiros.
- rational: Conjunto dos números fracionários.

A notação de dados também define as seguintes operações numéricas:

- sum x y: Retorna a soma dos números x e y.
- difference x y: Retorna a diferença entre os números x e y.
- product x y: Retorna o produtor de x e y.
- quotient x y: Retorna a divisão de x por y.
- negation x: Retorna o número x com sinal invertido.
- exponent x y: Retorna x elevado a potência y.
- sucessor x: Retorna $x + 1$.
- predecessor x: Retorna $x - 1$.

Valores Verdade

A notação de dados inclui o sorte truth-value para representar os valores booleanos, esse sorte é formado pelos indivíduos true e false e possui as seguintes operações:

- not x: Retorna a negação lógica de x.
- both x y: Retorna o resultado da operação lógica "AND" entre x e y.
- either x y: Retorna o resultado da operação lógica "OR" entre x e y.
- all(x_1,x_2,...): Retorna o sorte true se todos os parâmetros forem true.
- any(x_1,x_2,...): Retorna o sorte true se pelos menos um dos parâmetros for o sorte true.
- if x then y else z: Se x for o sorte true o valor desse operador será o do operando y, caso contrário o valor produzido pelo operado será o operando z.

Caracteres

A notação de dados define o sorte character que representa o conjunto de caracteres ASCII. As principais operações de manipulação de caracteres são:

- character of n: Retorna o caractere correspondente ao número natural n.
- code c: Devolve o número natural associado ao caractere c.

Listas

Listas são formadas por uma sequência de valores. O sorte empty-list define uma lista vazia. As listas possuem as seguintes operação pré-definidas:

- list of (i_1,i_2,...): Cria uma lista formada pela sequência de valores i_1, i_2,...
- head l: Retorna o primeiro elemento da lista l.
- tail l: Retorna a cauda da lista l.
- concatenation (l_1,l_2,...): Cria uma lista formada pela concatenação das listas recebidas l_1, l_2, etc.
- list [s]: Representa o sorte formado pelas listas formadas por elementos do sorte s.

Strings

A notação de dados define o sorte string como sendo as listas formadas por caracteres. São definidas as seguintes operações sobre strings:

- x ^ y: Retorna a string formada pela concatenação das strings x e y.
- lowercase x: Retorna uma string formada pela string x, convertendo as letras maiúsculas em minúsculas.

- uppercase x: Retorna uma string formada pela string x, convertendo as letras minúsculas em maiúsculas.

- decimal x: Retorna o natural equivalente a string x, que deve ser formada apenas por dígitos.

Conjuntos

Conjuntos são representados na notação de ações através do sorte set. O conjunto vazio é definido através do sorte empty-set. São definidas as seguintes operações sobre conjuntos:

- set of (x_1,x_2,..): Define um conjunto formados pelos elementos x_n.

- union (x_1, x_2,...): Retorna o conjunto formado pela união dos conjuntos x_n.

- intersection(x_1, x_2,...): Retorna o conjunto formado pela interseção dos conjuntos x_n.

- difference x y: Retorna o conjunto formado pela diferença entre x e y.

- x is in y: Retorna true se o elemento x está contido no conjunto y.

- x is included y: Retorna true se o conjunto x é um subconjunto de y.

- set [x] : Define o sorte dos conjunto formados por indivíduos do sorte x.

Mapeamentos

Mapeamentos são formados por associação entre indivíduos do sorte de domínio (element) e elementos do sorte range. O mapeamento vazio é representado pelo sorte empty-map. São definidas as seguintes operações que manipulam mapeamentos:

- map of e to r: Produz um mapeamento que associa o elemento do domínio e a imagem r.

- disjoint-union (m_1,m_2,...): Produz um mapeamento formado pela união dos mapeamentos m_1,m_2,..., sendo que um mesmo elemento do domínio não pode pertencer a mais de um desse mapeamentos.

- overlay (m_1,m_2,...): Similar a disjoint-union exceto que, nos casos em que um elemento de domínio estiver em mais de um dos mapeamentos, as associações do mapeamento m_1 prevalecerão sobre os mapeamentos m_2,m_3,etc, as associações de m_2 terão preferência sobre os mapeamentos m_3,..., e assim por diante.

- m at d: Retorna o elemento associado ao valor d no mapeamento m.

- map [d to e] : Identifica o sorte formado pelos mapeamentos que associam indivíduos do sorte d a indivíduos do sorte e.

Tuplas

As tuplas, identificadas pelo sorte tuple, são formadas por uma sequência de indivíduos, uma tupla vazia é identificada pelo sorte "()", os indivíduos do sorte component identificam as tuplas unitárias, tuplas de mais de um elemento são formadas através da operação de concatenação de tuplas "(t_1, t_2,...)". Além da operação de concatenação são definidas ainda as seguintes operações:

- $t^?$: Representa o sorte formado pela tupla vazia e pela tupla t.

- t^*: Representa o sorte formado por zero ou mais elementos do sorte t.

- t^+: Representa o sorte formado por um ou mais elementos do sorte t.

- t^n: Representa o sorte formado por n elementos do sorte t.

- count t: Retorna o tamanho da tupla.

- first t: Retorna o primeiro componente da tupla.

- rest t: Retorna uma tupla formada pela tupla t, excluindo-se o primeiro elemento.

- component# p t: Retorna o p-ésimo componente da tupla t.

4.2.2 Ações e Produtores

Ações e produtores representam a parte dinâmica da notação de ações. Eles são utilizados para modelagem das estruturas de controle e dos fluxos de informações dentro de um programa.

Uma ação é uma entidade que pode ser executada. Quando uma ação é executada, ela recebe informações do ambiente e, após sua execução, gera novas informações para o ambiente assim como um estado de término (outcome) que indica o resultado da execução.

Os principais estados de término são:

- completing: Indica o término normal da execução da ação.
- failing: Indica o término anormal da execução da ação.
- diverging: Indica a impossibilidade da conclusão da execução da ação.
- escaping: Indica a ocorrência de uma sequência de escape. Uma descrição mais detalhada desse estado será visto mais adiante.

As ações podem ser divididas em ações primitivas e ações combinadas, que são formadas por uma ou mais subações unidas através de combinadores de ações.

Os produtores (yielders) são entidades que são avaliadas dinamicamente durante a execução de uma ação. Para serem avaliados, os produtores necessitam das informações contidas no ambiente de execução. Os produtores são formados pelos sortes e operadores definidos pela notação de dados, que independem das informações do ambiente de execução, e por operadores especiais dependentes das informações do ambiente.

As ações e produtores são divididos em várias facetas, segundo o tipo de informação eles manipulam. Cada faceta identifica as ações, produtores e estruturas de dados encarregadas de modelar um determinado conceito existente em linguagens de programação.

As próximas seções descreverão as características de cada faceta e suas principais ações primitivas, combinadores e produtores.

Uma especificação formal das facetas da notação de ações pode ser encontrado no Apêndice B, a descrição formal da execução de uma ação, pode ser encontrada no Apêndice D.

Faceta Básica

A faceta básica agrupa ações relacionadas com o fluxo de controle de um programa. As principais ações básicas primitivas são:

- complete: Essa ação ao ser executada termina normalmente (outcome completing).
- fail: Essa ação termina produzindo uma mensagem de erro (outcome failing).
- diverge: Essa devolve o estado diverging ao ser executada, indicando o não término da execução da ação.
- escape: Gera uma instrução de escape que transferirá o fluxo de execução para a o combinador trap mais próximo.
- commit: Compromete a execução da ação, o comprometimento da execução de uma ação como veremos mais adiante influenciará a semântica do combinador or.
- unfold: Para uma descrição dessa ação vide a descrição do combinador unfolding.

Os principais combinadores básicos de ações são:

- a_1 and a_2: Executa as ações a_1 e a_2 intercaladamente.
- a_1 and then a_2: Executa sequencialmente as ações a_1 e a_2.

- a_1 or a_2: Executa uma das ações a_1 ou a_2 ao acaso e se a ação escolhida terminar normalmente o combinador encerra a execução produzindo o estado completing. Por outro lado, se a ação escolhida retornar o estado failing então:
  - Se a execução da ação escolhida estiver comprometida, então a execução do combinador or terminará com uma mensagem de erro.
  - Caso contrário, o combinador tentará executar a ação alternativa.
- a_1 trap a_2: Executa a ação a_1, e se durante a sua execução, for gerada uma instrução de escape. A execução da ação a_1 será interrompida e a ação a_2, será executada. Caso contrário, a ação a_2 será ignorada.
- unfoldind a: Executa a ação a substituindo toda ocorrência da ação unfold dentro da ação a por unfolding a.

Observações:

1. As ações complete, fail, escape, diverge, commit são ações indivisiveis, ou seja, a sua execução não pode ser divididas em duas ou mais subetapas.
2. Ações combinadas complexas podem utilizar uma notação de barras verticais para simplificar a legibilidade e definir o agrupamento de subações. Nessa notação uma expressão na forma:

```
| x
op
| y
```

equivale a expressão:

```
(x) op (y).
```

Como exemplo dessa notação, a ação:

```
| | complete
| or
| | fail
and then
| | escape
| or
| | commit.
```

é equivalente a ação:

((complete) or (fail)) and then ((escape) or (commit))

Exemplo 4.1

1. complete and then fail

   Durante a execução dessa ação, a primeira ação (complete) será executada, terminando satisfatoriamente. Após a sua execução, a segunda ação será executada, fazendo com que a ação gere um erro de execução.

2. complete and fail

   Ao contrário da ação anterior, o combinador and não define a ordem de execução das ações. Se a ação a ser executada for a complete, então após sua execução a ação fail será executada, produzindo um outcome failing. Caso contrário, se a ação fail for executada a execução da ação combinada produzirá um outcome failing e a ação complete não vai ser executada.

3. fail and then complete

   Durante a execução dessa ação, a ação fail será executada, gerando um outcome failing. Como a primeira ação não foi executada com sucesso, o combinador and then não executará a segunda ação, produzindo um término anormal.

4. complete or fail

   Durante a execução dessa ação, uma das duas ações será escolhida para ser executada. Se a ação escolhida for a primeira, que terminará normalmente, então o combinador termina a sua execução. Se, por outro lado, a ação executada for a ação fail, que produz um outcome failing, o combinador or executará a ação complete.

5. complete or (commit and then fail)

De forma similar a ação anterior, uma das ações será executada. Se a ação escolhida for a primeira, então a ação terminará normalmente. Entretanto, se a segunda ação (commit and then fail) for executada, ela retornará um erro para o combinador que ficará impossibilitado de executar a ação alternativa devido a execução da ação commit retornando um estado de erro.

6. unfolding
   | complete or unfold

A ação unfolding ao ser executada executa a subação correspondente: complete or unfold. Se a ação complete for executada, a subação e o combinador unfolding terminarão. Se por outro lado a ação unfold for executada, a ação unfolding (complete or unfold) será executada novamente.

7. | escape and then fail
   trap
   | complete

Durante a execução dessa ação a subação (escape and then fail) será executada, nesse caso, a ação escape será executada transferindo o fluxo de controle para a subação complete, ignorando dessa o fluxo de instruções estabelecido pelo combinador and then.

□

Faceta Funcional

A faceta funcional engloba as ações que manipulam informações transitórias, essas informações representam dados intermediários temporários que são usados pelas próximas ações do fluxo de execução.

As informações transitórias são definidas formalmente como indivíduos do sorte **data**, definido como tuplas formadas de zero ou mais indivíduos do sorte **datum** (**datum***). O sorte **datum** abrange os sortes numéricos, strings, truth-values, entre outros.

As principais ações funcionais primitivas são:

- **give** e: Essa ação ao ser executada, devolve o resultado da expressão e como informação transitória.
- **escape with** e: Essa ação ao ser executada, transfere o fluxo de controle para a combinador **trap** mais próximo passando como dado temporário o valor da expressão e.
- **regive**: Essa ação repassa todas as informações transitórias que ela receber.
- **check** e: Essa ação avalia a expressão e e termina normalmente se o seu resultado for **true**, falhando caso contrário.

O principal combinador funcional é o combinador **then**. A ação a_1 **then** a_2, ao ser executada passa os transitórios recebidos pelo combinador para a ação a_1. Após a execução da ação a_1 as informações transitórias produzidas pela ação são fornecidas para a execução da ação a_2 que produzirá as informação transitórias que serão devolvidas como resultado da execução do combinador.

Observações:

1. As ações básicas primitivas não produzem dados transitórios
2. Os combinadores básicos lidam com informação transitória repassando a informação transitória recebida para as suas subações, que serão executadas e os transitórios produzidos por estas são concatenados e devolvidos como resultado funcional da ação combinada.

A faceta funcional define os seguintes produtores:

- given d: Retorna a informação transitória recebida pela ação.
- given $d\#n$: Retorna a n-ésima informação transitória recebida pela ação.

Em ambos os casos, o dado retornado deve ser um subsorte do sorte d ou ação que estiver executando falhará.

Exemplo 4.2

1. give 2

   Essa ação fornece o sorte "2" como informação transitória.

2. give 1 and then give 2

   Durante a execução dessa ação. A primeira ação give 1 é executada fornecendo como informação transitória o sorte 1. A próxima instrução give 2 é então executada fornecendo o sorte 2 como informação transitória. Após a execução das duas subações, o combinador and then irá produzir como transitórios a tupla formada pela concatenação dos transitórios produzidos pelas subações, ou seja (1,2).

3. give 1 then give 2

   Durante a execução, a primeira ação é executada produzindo a informação transitória 1. Após a execução da primeira ação, o combinador then executará a segunda ação, passando as informações transitórias produzidas pela primeira ação. A segunda ação fornecerá a informação transitória 2 que será o resultado da ação combinada.

4. ```
   | give 1 and give 2
   then
   | give sum (the given integer#1) (the given integer#2)
   ```

   O combinador then executará a primeira subação (give 1 and give 2), que devolverá a tupla (1,2) como informação transitória. As informações transitórias são passadas para a segunda subação (give sum (the given integer#1) (the given integer#2)), que utilizará as informações transitórias (através dos produtores the given integer#1 e the given integer#2) como parâmetros para operação sum, que fornecerá o valor retornado pela ação give.
   ```

5.
```
| give true
then
| give the given integer
```

□

Faceta Declarativa

As ações da faceta declarativa são responsáveis pela manipulação das informações de escopo (bindings) . As informações de escopo identificadores (tokens), normalmente definidos como strings, a valores pertencentes ao sorte bindable, responsável por definir quais valores podem estar associados aos tokens.

A descrição do sorte bindable é deixada em branco pela notação de ações, cabendo à especificação da linguagem definir que valores podem estar associados aos tokens.

A notação de ações define um conjunto de ações primitivas e combinadores para manipulação do escopo de um programa. As principais ações declarativas são:

- bind t to e: Produz uma associação entre o token t e o valor da expressão e que, obrigatoriamente, deve ser um subsorte do sorte bindable.
- rebind: Devolve todas as associações recebidas pela ação.

Os principais combinadores que manipulam o escopo de um programa são:

- a_1 hence a_2: Executa as ações a_1 e a_2 sequencialmente.

  As associações recebidas pelo combinador são passadas para a ação a_1. A ação a_2 receberá as associações produzidas pela ação a_1 ao ser executada e as associações que ela produzir formarão o resultado da ação combinada.
- a_1 before a_2: Executa as ações a_1 e a_2 sequencialmente.

As associações recebidas pelo combinador são passadas para a ação a_1. As associações produzidas por essa ação são adicionadas às recebidas pelo combinador e passadas para a ação a_2, que será executada. As associações produzidas pela ação combinada são formadas pela união das associações produzidas pelas subações a_1 e a_2.

- a_1 moreover a_2: Executa as ações a_1 e a_2 sequencialmente.

  As associações recebidas pelo combinador são passadas para as ações a_1 e a_2. Após a execução das duas ações, o combinador produzirá a união das associações produzidas pelas subações.

- furthermore a_1: Executa a ação a_1 com as associações recebidas pelo combinador e retorna a união das associações produzidas pela ação a_1 e as associações recebidas pelo combinador.

Observações:

1. As ações primitivas básicas e funcionais não produzem nenhuma associação.

2. Os combinadores declarativos tratam as informações transitórias de forma análoga aos combinadores básicos, ou seja, as informações transitórias recebidas são passadas para as suas subações e o combinador produzirá a união das informações transitórias produzidas por estas.

3. Os combinadores básicos e funcionais processam associações repassando as informações de escopo recebidas para as subações e produzindo a união das associações produzidas por estas.

A faceta declarativa define o produtor the d bound to t, que retorna o dado que estiver associado ao token t nos associações recebidas pelo produtor. Se não houver nenhuma informação associada ou esta não for um subsorte do sorte d, ação que estiver executada falhará.

Exemplo 4.3

1. bind "x" to 10

   O resultado da execução dessa ação é a associação entre o token "x" e o sorte 10, nesse caso estamos considerando que sorte 10, é um subsorte do sorte bindable. Caso contrário a ação terminaria, produzindo um outcome failing.

2. | bind "x" to 10
   before
   | bind "y" to 20

   Ao ser executada, o combinador before executará a primeira ação que associará o token "x" ao sorte 10. Essa associação é adicionada às associações recebidas e o resultado é passado para a segunda ação, que produzirá uma associação entre o token "y" e o sorte 20. Após a execução do combinador, este unirá as associações produzidas pelas suas subações e retornará o valor resultante como as associações produzidas pelo combinador.

3. | bind "x" to 10
   hence
   | give the integer bound to "x"

   Durante a execução dessa ação, a primeira ação será executada produzindo uma associação. Essa associação será passada para a segunda ação que consultará o valor associado, através do produtor the _ bound to _, fornecendo o valor retornado como informação transitória. O resultado da ação combinada são as informações transitórias produzidas pelas duas subações (2) e as associações produzidas pela segunda subação (nenhuma).

□

Faceta Imperativa

As ações da faceta imperativa são responsáveis pela manipulação da memória (storage) do programa. A memória é formado por um conjunto (virtualmente) infinito de células de memória. Em um determinado momento, cada célula de memória pode estar alocada ou não, se essa célula estiver alocada ela armazenará um valor que poderá ser consultado ou alterado pelas ações.

Os valores que podem ser armazenados pelas células são subsortes do sorte storable, que deve ser definido pela especificação da linguagem de forma a se adequar às características da linguagem. Um outro valor que pode ser armazenado na memória é o uninitialized que indica que nenhum dado válido esta armazenado na memória.

As seguintes ações primitivas são responsáveis pela manipulação da memória:

- store e in c: Essa ação armazena o valor da expressão e na célula de memória referenciada pela expressão c. O valor da expressão e deve ser do sorte storable e a célula referenciada por c deve estar alocada, se uma dessas condições não forem satisfeitas a execução da ação falhará.

- deallocate c: Essa ação desalocará a célula c de memória, que deve estar alocada caso contrário a ação falhará.

- allocate a cell[1]: Essa ação, ao ser executada, escolhe uma célula não alocada da memória, muda o seu estado para alocada e retorna o seu valor como informação transitória.

A faceta imperativa define ainda o produtor the d stored in c, esse produtor devolve o dado armazenado na célula retornada pela expressão c. Caso a expressão c não retorne o valor de uma célula alocada ou o valor armazenado não seja subsorte de d, ação que estiver sendo executada falhará.

A execução de uma ação imperativa comprometerá a execução da ação da mesma forma que a execução da ação commit.

[1]A ação primitiva allocate a cell é classificada em [Mos92] como uma ação híbrida, pois altera a memória corrente e produz uma informação transitória, entretanto resolvemos inclui-la como ação imperativa para simplificação do texto.

Exemplo 4.4

1. | allocate a cell
   then
   | store 10 in the given cell.

   A execução dessa ação se dá da seguinte forma: a ação allocate a cell, alocará uma posição de memória livre e devolverá o valor alocado nas informações transitórias, que serão repassadas pelo combinador then para a segunda ação que recuperará o valor transitório (produtor given) e armazenará o sorte 10 na posição de memória.

□

Faceta Reflexiva

A faceta reflexiva é responsável pela manipulação de abstrações. Um abstração é capaz de armazenar ações que poderão ser executadas posteriormente.

A faceta reflexiva define um sorte de nome abstraction para representar as abstrações. As principais operações dessa faceta são:

- abstraction of a: Retorna uma abstração associada a ação a.
- closure ab: Retorna uma abstração formada pela abstração ab, que armazena as informações de escopo corrente que serão utilizadas na execução da ação incorporada.

A principal ação existente na faceta reflexiva é a ação enact e. Ao ser executada, essa ação avaliará a expressão e, que deve retornar uma abstração, e em seguida executará a ação encapsulada pela abstração. As informações recebidas pela ação são passadas para a ação encapsulada que produzirá o resultado da execução da ação enact.

Exemplo 4.5

1. abstraction of
   | give 10.

Essa expressão retorna uma abstração que incorpora a ação: give 10.

2.
```
| give abstration of
| | give 10.
then
| enact the given abstration.
```

Essa ação ao ser executada, executará a ação give que produzirá um dado transitório contendo uma abstração. Esse transitório é repassado para a ação enact, que recuperará o valor transitório e executará a ação give 10 representada pela abstração.

□

Faceta Comunicativa

A faceta comunicativa é responsável pela modelagem de agentes e mensagens. Agentes são entidades responsáveis pela execução das ações. Esses agentes podem ser criados dinamicamente e se comunicam de forma assíncrona através de passagens de mensagens.

As principais ações definidas pela faceta comunicativa são:

- subordinate an agent: Essa ação cria um novo agente. O comportamento inicial desse novo agente é esperar uma mensagem contendo uma ação que deverá ser executada pelo agente. Após ser executado, a ação retorna o identificador desse agente como informação transitória.
- send m: Envia a mensagem m.
- receive m: Recebe uma mensagem definida por m e a retorna como informação transitória.

As informações que podem ser enviadas como mensagens são definidas pelo sorte message. As seguintes operações são definidas para mensagens:

- d [from a]: Define as mensagens de sorte d enviadas pelo agente a.

- d [to a]: Define as mensagens de sorte d enviadas para o agente a.

4.3 Descrição de Linguagens de Programação

A descrição de uma linguagem de programação em semântica de ações é divida em três partes: sintaxe abstrata, funções semânticas e entidades semânticas.

4.3.1 Sintaxe Abstrata

A parte de sintaxe abstrata define a estrutura sintática dos programa válidos escritos na linguagem especificada, utilizando para isso uma notação similar à BNF [ASU86]. Essa descrição deve se preocupar apenas em indicar uma representação abstrata dos programas, ignorando aspectos de implementação da gramática tais como: reconhecimento de tokens, eliminação de conflitos, etc.

A descrição da sintaxe abstrata de um programa é feita através do sorte syntax-tree, definido na notação de dados. Esse sorte é um supersorte do sorte string e possui os seguintes operadores:

- $[\![\ a_1\ a_2\ ...\ ]\!]$: Define uma sequência formada pelas árvores sintáticas a_1, a_2, ...
- x^*: Define uma árvore sintática formada pela sequência de 0 ou mais ocorrências da árvore x.
- x^+: Define uma árvore sintática formada pela sequência de 1 ou mais ocorrências da árvore x.
- $x^?$: Define uma árvore sintática formada por 0 ou 1 ocorrências da árvore x.

Exemplo 4.6

Programa = $[\![$ Declaração "in" Comando $]\!]$.

A especificação acima define que um programa é definido pela árvore sintática produzida por uma declaração seguida da string "`in`", seguida por um comando.

□

4.3.2 Funções Semânticas

As funções semânticas fornecem o significado dos programas da linguagem especificada e são formadas por operadores que recebem como parâmetros a descrição sintática das frases do programa e devolvem o seu significado. O significado de uma frase sintática é dado através da notação de ações. A ação que fornece o significado de um programa em uma linguagem especificada é denominada uma ação-programa.

Exemplo 4.7

A seguinte função semântica fornece o significado dos programas de uma linguagem:

(1) run $[\![$ d:Declaração "`in`" c:Comando $]\!]$ =
    | elaborate d
    hence
    | execute c.

como podemos observar, a semântica de um programa é expressa através da notação de ações e é definida como a combinação das semânticas de suas partes componentes, obtidas através de outras funções semânticas.

□

4.3.3 Entidades Semânticas

As entidades semânticas de uma linguagem definem sortes e operações complementares que auxiliarão na descrição da linguagem de programação. Normalmente, as entidades semântica de uma descrição de linguagem de programação definem pelo menos os sortes deixados em aberto pela notação de ações: bindable e storable.

4.4 Descrição de uma Linguagem de Programação

Para exemplificar a utilização de semântica de ações definiremos a semântica de uma pequena linguagem de programação imperativa de denominaremos de SIMPLE.

4.4.1 Descrição Informal

Um programa na linguagem SIMPLE é formado por um conjunto de declarações, sequidas por um comando que deve ser executado.

A linguagem de programação SIMPLEpossui dois tipos de declarações:

1. `var` x `:=` exp - Que define uma variável de nome x com valor inicial exp. Em SIMPLE, uma variável pode armazenar valores inteiros ou lógicos indiscriminadamente.
2. `proc` p `:=` c `end` - Que define um procedimento de nome p que executará o comando c ao ser invocado.

Os comandos definidos pela linguagem SIMPLE são os seguintes:

1. v `:=` e - Atribui á variável v o valor da expressão e.
2. `call` p - Executa o procedimento p.
3. `if` e `then` c_1 `else` c_2 - Se a expressão e retornar o valor `true`, então o comando c_1 será executado. Caso contrário o comando c_2 será executado.
4. `while` e `do` c `end` - Executa o comando c até que a expressão e possua um valor falso.

5. `local` d `in` c `end` - Produz as declarações d válidas durante a execução do comando c.

Dois ou mais comandos podem ser agrupados através do operador "`;`".

As expressões de SIMPLE podem ser:

1. Uma constante inteira ou lógica.
2. Um nome de variável.
3. A aplicação de um operador pré-definido.

4.4.2 Sintaxe Abstrata

A seguir apresentamos a sintaxe dos programas da linguagem especificada.

Programas

Program = ⟦ Declaration "`in`" Command ⟧ .

Declarações

Declaration =
⟦ "`var`" Identifier "`:=`" Expression ⟧ **|**
⟦ "`proc`" Identifier "`:=`" Command "`end`"⟧ **|**
⟦ Declaration "`;`" Declaration ⟧.

Comandos

Command =
⟦ Identifier "`:=`" Expression ⟧ **|**
⟦ "`call`" Identifier ⟧ **|**
⟦ "`if`" Expression "`then`" Command "`else`" Command "`end`" ⟧ **|**
⟦ "`while`" Expression "`do`" Command "`end`" ⟧ **|**
⟦ "`local`" Declaration "`in`" Command "`end`" ⟧ **|**
⟦ Command "`;`" Command ⟧.

Expressões

Expression =
⟦ Identifier ⟧ |
⟦ Number ⟧ |
⟦ Expression Op Expression ⟧ |
⟦ Op Expression ⟧ |
⟦ "(" Expression ")" ⟧.

Outras declarações

Op =
⟦ "+" ⟧ | ⟦ "-" ⟧ | ⟦ "*" ⟧ | ⟦ "/" ⟧ | ⟦ "=" ⟧ |
⟦ "<>" ⟧ | ⟦ ">" ⟧ | ⟦ ">=" ⟧ | ⟦ "<" ⟧ | ⟦ "<=" ⟧ |
⟦ "and" ⟧ | ⟦ "or" ⟧ | ⟦ "not" ⟧.

Identifier = ⟨ alpha ⟩$^+$.

Number = ⟨ digit ⟩$^+$.

4.4.3 Funções Semânticas

Executando Programas

- run _ :: Program → action.

(1) run ⟦ d:Declaration "in" c:Command ⟧ =
elaborate d hence execute c.

Elaborando Declarações

- elaborate _ :: Declaration → action.

(1) elaborare ⟦ "var" i:Identifier ":=" e:Expression ⟧ =
| allocate a cell and evaluate e
then
| | bind i to the given cell#1
| and
| | store the given value#2 in the given cell#1.

(2) elaborate ⟦ "`proc`" i:Identifier "`:=`" c:Command "`end`"⟧ =
recusively bind i to abstraction of (execute c).

(3) elaborate ⟦ d_1:Declaration "`;`" d_2:Declaration ⟧ =
elaborate d_1 before elaborate d_2.

Executando Comandos

- execute _ :: Command → action.

(1) execute ⟦ i:Identifier "`:=`" e:Expression ⟧ =
| evaluate e
then
| store the given integer in the cell bound to i.

(2) execute ⟦ "`call`" i:Identifier ⟧ = enact the abstration bound to i.

(3) execute ⟦ "`if`" e:Expression "`then`" c_1:Command "`else`" c_1:Command "`end`" ⟧ =
| evaluate e
then
| | check the given truth-value then execute c_1
| or
| | check not the given truth-value then execute c_2.

(4) execute ⟦ "`while`" "`(`" e:Expression "`)`" "`do`" c:Command ⟧ =
unfolding
| | evaluate e .
| then
| | | | check the given value
| | | then
| | | | execute c then unfold
| | or
| | | check not the given truth-value.

(5) execute ⟦ "`local`" d:Declaration "`in`" c:Command "`end`" ⟧ =
elaborate d hence execute c.

(6) execute ⟦ c_1:Command "`;`" c_2:Command ⟧ =
execute c_1 and then execute c_2.

Avaliando Expressões

- evaluate _ :: Expression $\rightarrow$ action.

(1) evaluate ⟦ i:Identifier ⟧ = give the value stored in the cell bound to i.

(2) evaluate ⟦ n:Number ⟧ = give number of n.

(3) evaluate ⟦ e_1:Expression o:Op o:Expression ⟧ =
  | evaluate e_1 and evaluate e_2
  then
  | make-binary-op o.

(4) evaluate ⟦ o:Op e:Expression ⟧ =
  evaluate e then make-unary-op o.

(5) evaluate ⟦ "(" e:Expression ")" ⟧ = evaluate e.

Operadores Binários

- make-binary-op _ :: Op $\rightarrow$ action.

(1) make-binary-op ⟦ "+" ⟧ = give sum (the given integer#1) (the given integer#2).

(2) make-binary-op ⟦ "-" ⟧ = give difference (the given integer#1) (the given integer#2).

(3) make-binary-op ⟦ "*" ⟧ = give product (the given integer#1) (the given integer#2).

(4) make-binary-op ⟦ "/" ⟧ = give division (the given integer#1) (the given integer#2).

(5) make-binary-op ⟦ ">" ⟧ =
  give (the given integer#1) is greater than (the given integer#2).

(6) make-binary-op ⟦ ">=" ⟧ =
  give (the given integer#1) is greater-equal than (the given integer#2).

(7) make-binary-op ⟦ "<" ⟧ =
  give (the given integer#1) is less than (the given integer#2).

(8) make-binary-op ⟦ "<=" ⟧ =
  give (the given integer#1) is less-equal than (the given integer#2).

(9) make-binary-op ⟦ "=" ⟧ = give (the given value#1) is (the given value#2).

(10) make-binary-op ⟦ "<>" ⟧ = give not (the given value#1) is (the given value#2).

(11) make-binary-op ⟦ “and” ⟧ = give both (the given truth-value#1) (the given truth-value#2).

(12) make-binary-op ⟦ “or” ⟧ = give either (the given truth-value#1) (the given truth-value#2).

Operadores Unários

- make-unary-op _ :: Op → action.

(1) make-unary-op ⟦ “-” ⟧ = give negation of the given integer#1.

(2) make-unary-op ⟦ “not” ⟧ = give not the given truth-value#1.

4.4.4 Entidades Semânticas

introduces: bindable, storable, value.

(1) bindable = cell | abstration.

(2) storable = integer.

(3) value = integer | truth-value.

As entidades semânticas da linguagem especificada definem que as células de memória são e abstrações as únicas entidades que podem ser associadas à identificadores (definição do sorte bindable). Somente os inteiros podem ser armazenados na memória (definição do sorte storable) e define um sorte de nome value que será utilizado pela especificação.

4.4.5 Um Exemplo de Uso da Especificação

Para exemplificar a utilização da especificação vista nessa seção, definiremos o significado do seguinte programa em Simple:

```
  var x := 0
in
  x := x + 1
```

O significado desse programa é obtido através do operador `run _`, que recebe a árvore sintática do programa e devolve o seu significado, expresso através de uma ação-programa. A aplicação do operador `run _`, recebendo como parâmetro o programa acima, produzirá a seguinte ação:

```
| | | allocate a cell
| | and
| | | give 0
| then
| | | bind "x" to the given cell # 1
| | and
| | | store the given value # 2 in the given cell # 1
hence
| | | | give the value stored in the cell bound to "x"
| | | and
| | | | give 1
| | then
| | | give the sum (the given value # 1) (the given value # 2)
| then
| | store the given value in the cell bound to "x"
```

que representará o significado do programa.

Especificações de linguagens de programação como as mostradas nesse exemplo podem ser implementadas manualmente com relativa facilidade através de linguagens de programação convencionais. Uma outra abordagem de implementação, que utilizaremos nos próximos capítulos, se aproveita do fato de especificações de linguagens de programação serem especificações algébricas unificadas. Essa característica possibilita o uso do tradutor de especificações algébricas descrito no Capítulo 3 para a geração automática de implementações para descrições de linguagens de programação em semântica de ações.

Capítulo 5

O Sistema Abaco

O objetivo desse capítulo é apresentar o sistema Abaco, um sistema gerador de compiladores baseado em semântica de ações.

Iniciaremos apresentando as características básicas de um compilador baseado em semântica de ações e alguns sistemas resultantes de trabalhos de pesquisa na área de geração automática de compiladores baseada em semântica de ações.

Na sequência apresentaremos o sistema Abaco, que tem como base o método de tradução de especificações algébricas unificadas descrito no Capítulo 3. Finalmente, citamos as principais vantagens obtidas com a utilização do sistema proposto.

5.1 Compiladores Baseados em Semântica de Ações

O esquema geral de um compilador baseado em semântica de ações para uma linguagem $\mathcal{L}$ é mostrado na Figura 5.1. Nessa figura, o compilador é formado pelos seguintes módulos:

- Um parser que recebe um programa fonte de $\mathcal{L}$ e produz a árvore sintática abstrata correspondente ao fonte. Esse módulo normalmente é produzido com o uso de ferramentas como Lex [LS79] e Yacc [Joh79] a partir da sintaxe abstrata da linguagem.

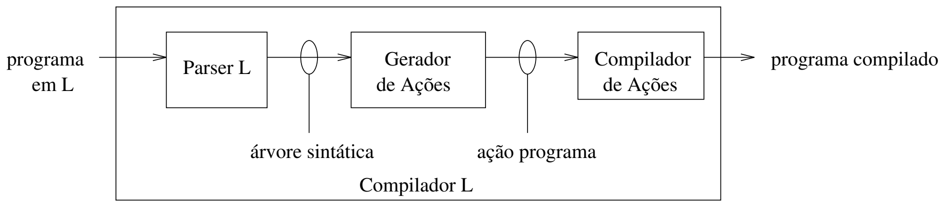

Figura 5.1: Estrutura de um Compilador Baseado em Semântica de Ações

- Um gerador de ações, responsável por receber a árvore sintática abstrata produzida pelo parser e gerar a ação programa que forneça o significado do programa. Esse módulo é produzido a partir das funções semânticas da linguagem através de ferramentas específicas definidas por um gerador de compiladores para notação de ações.

- Um compilador de ações, responsável por gerar o programa compilado final a partir da ação programa gerada pelo gerador de ações. Esse módulo geralmente é independente da especificação da linguagem de programação e deve ser capaz de realizar, entre outras coisas, passos de otimização com o objetivo de aumentar a eficiência do código gerado pelo compilador.

5.2 ACTRESS

O sistema gerador de compiladores ACTRESS foi desenvolvido por Moura, Brown, e Watt [BMW92, Bro97, Mou93]. O sistema ACTRESS, descrito na Figura 5.2, define as seguintes ferramentas para geração de compiladores a partir de semântica de ações:

- *Actioneer Generator* (AG): responsável por receber a descrição das funções semânticas da linguagem e produzir o acionador de ações para a linguagem (equivalente ao gerador de ações descrito na Seção 5.1).

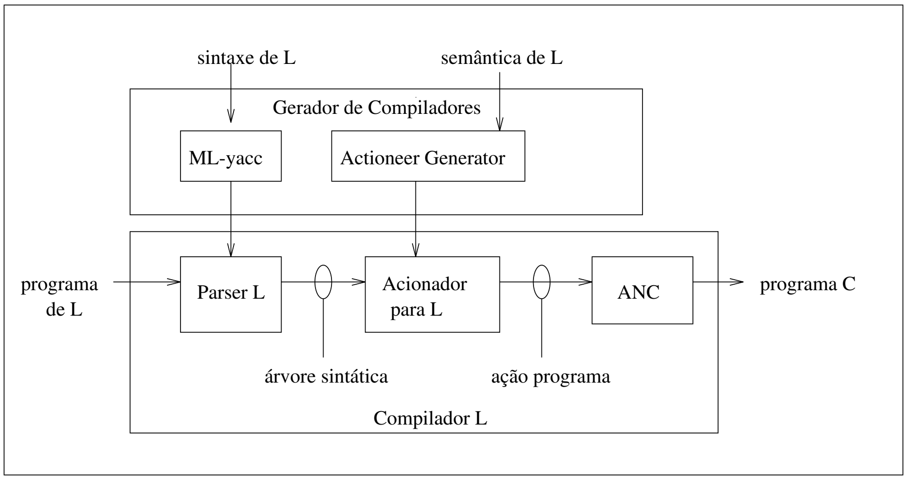

Figura 5.2: Arquitetura de Actress

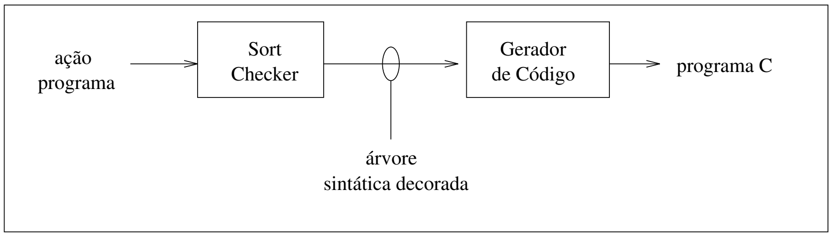

Figura 5.3: Estrutura do ANC (back end)

- *Action Notation Compiler* (ANC): O ANC é equivalente ao compilador de ações visto na Seção 5.1, a estrutura do ANC, como podemos observar na Figura 5.3, é composta por um analisador de ações e um gerador de código. O analisador de ações validará a ação programa fornecida e produzirá uma ação decorada que conterá informações úteis para melhorar a geração de código, o seu funcionamento é detalhado em [Bro97]. O gerador de código receberá a ação decorada e produzirá um programa C, realizando um conjunto de transformações, descritas em [Mou93], que otimizarão a ação, permitindo a obtenção de um código gerado de melhor qualidade.

5.3 Cantor

O sistema CANTOR [Pal92] formado por duas ferramentas principais:

- Um *pré-processador*, escrito em PERL [WS91], responsável por processar especificações de linguagens em semântica de ações, escritas em LaTeX [Lam85], produzindo geradores de ações escritos em SCHEME [SF90].
- Um *compilador de ações*, formado através de dois módulos:
    1. um programa que traduz a ações em código assembly Pseudo-SPARC.
    2. um compilador de código Pseudo-SPARC, responsável pela geração do executável final.

A principal contribuição de Cantor foi a definição de uma prova formal da corretude do compilador de ações a partir de uma semântica natural da notação de ações dada.

5.4 Oasis

O sistema OASIS (Optimizing Action-based Semantic Implementation System), gera compiladores para um subconjunto da notação de ações [Ørb93]. O sistema é formado por:

1. Um *pré-processador* escrito em PERL que recebe as funções semânticas de uma linguagem e produz o gerador de ações para a linguagem especificada.
2. Um *analisador e otimizador de ações*, que recebe a ação programa e gera um código diretamente para linguagem assembly realizando, entre outras, as seguintes análises e otimizações na ação programa recebida:
    - propagação de constantes
    - analise do fluxo de execução

- alocação de variáveis
- análise de término da ação

Os resultados obtidos em testes com esse sistema mostraram que as técnicas de otimização usadas conseguiram gerar executáveis com performance comparável aos gerados por compiladores produzidos manualmente.

5.5 O Sistema ABACO

Nessa seção descreveremos a arquitetura do sistema ABACO (Algebraic Based Action COmpiler), um gerador de compiladores baseado em semântica de ações, proposto nessa dissertação. Esse sistema tem como principal característica a utilização do método de tradução de especificações algébricas unificadas, descrito no Capítulo 3, para auxiliar a prototipação de linguagens de programação.

A estrutura de um compilador para uma linguagem de programação $\mathcal{L}$, gerado pelo sistema ABACO, é mostrada na Figura 5.4. Como podemos ver nessa figura, o compilador gerado é formado pelos seguintes módulos:

- Um *parser* para a linguagem $\mathcal{L}$: Responsável por produzir a árvore sintática abstrata a partir de um programa fonte em $\mathcal{L}$. Esse módulo é gerado através de ferramentas como LEX e YACC.

- Um *gerador de ações*: Responsável por definir o significado de um programa em $\mathcal{L}$ a partir de sua sintaxe abstrata, utilizando a notação de ações.

- Um *simplificador de ações*: Responsável por converter o significado do programa, expresso através da notação de ações, para uma forma mais simples que possa ser compreendida pelo gerador de código.

- Um *compilador de ações*: Responsável por produzir uma primeira versão do código compilado.

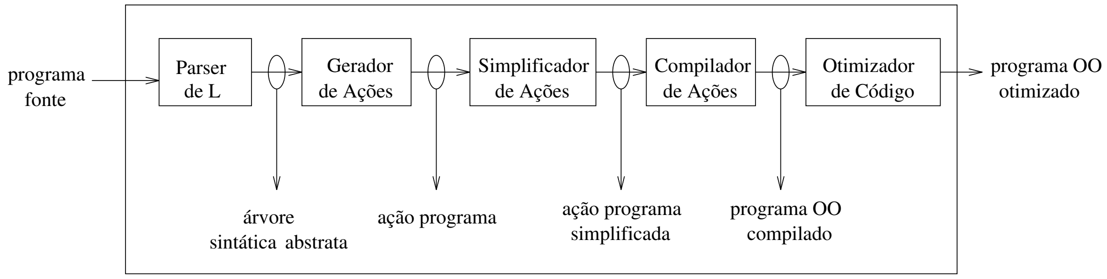

Figura 5.4: Compilador da Linguagem $\mathcal{L}$

- Um *otimizador de código*: Responsável pela realização de otimizações de baixo nível no código produzido pelo compilador de ações.

Todos esse módulos necessitam para serem implementados de uma biblioteca de apoio, denominada biblioteca auxiliar estática de $\mathcal{L}$. Essa biblioteca é produzida a partir da tradução orientada a objetos das seguintes especificações algébricas unificadas:

- Especificação da notação de ações, responsável por definir ações, produtores e dados existentes em semântica de ações.

- Especificação do simplificador de ações, responsável por definir regras de simplificação e otimização para a notação de ações.

- Especificação da linguagem de programação, que descreverá as características da linguagem $\mathcal{L}$.

A biblioteca auxiliar estática implementa a descrição da linguagem $\mathcal{L}$, juntamente com toda a notação de ações e com o simplificador de ações, gerando os módulos responsáveis pela geração e simplificação de ações. O esquema de produção da biblioteca auxiliar estática é mostrado na Figura 5.5.

As implementações do otimizador de código, do compilador de ações e do simplificador de ações, são descritas detalhadamente no Capítulo 6.

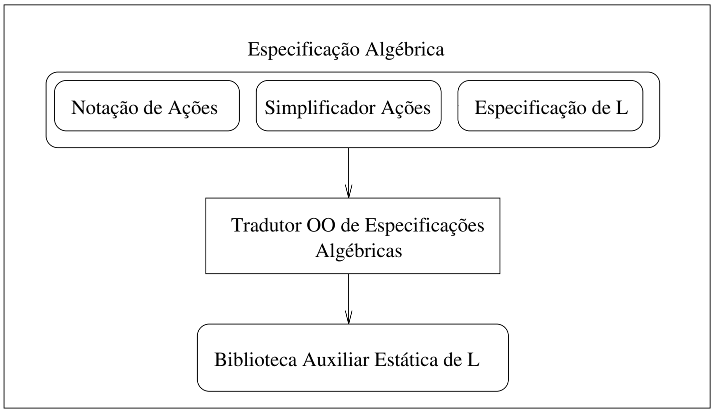

Figura 5.5: Geração da Biblioteca Auxiliar Estática de $\mathcal{L}$

O código produzido pelo compilador gerado pelo sistema ABACO é escrito na linguagem C++. Para produção do executável final, ele deve ser compilado e ligado a um conjunto de bibliotecas de apoio. Uma dessas bibliotecas, denominada biblioteca auxiliar dinâmica de $\mathcal{L}$, conterá a implementação das estruturas de dados definidas pela especificação de $\mathcal{L}$ que serão necessárias durante a execução de um programa em $\mathcal{L}$. A geração dessa biblioteca dá-se, como podemos ver na Figura 5.6, através da tradução das especificações da notação de dados de semântica de ações e das entidades semânticas da especificação de $\mathcal{L}$.

A principal vantagem que esperamos obter com o uso do sistema ABACO será o aumento de flexibilidade na descrição de linguagens de programação. Esse aumento dá-se pela facilidade de estendermos a notação de ações, definindo novas ações e estruturas de dados que poderão ser utilizadas automaticamente por novas especificações e protótipos.

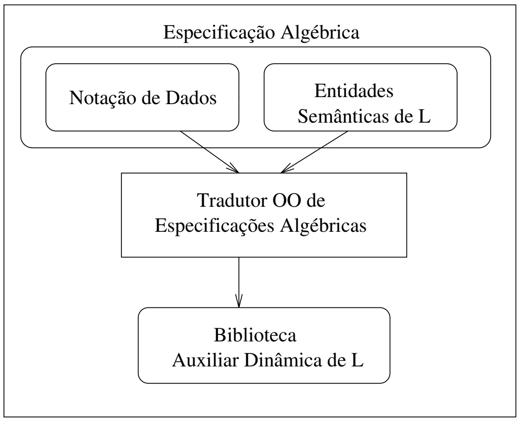

Figura 5.6: Geração da Biblioteca Auxiliar Dinâmica de $\mathcal{L}$

Capítulo 6

Um Tradutor para a Notação de Ações

Nesse capítulo descreveremos o processo de geração de código para a notação de ações, que constitui uma das principais ferramentas que compõe o sistema ABACO, descrito no capítulo anterior.

O tradutor de ações que descreveremos nesse capítulo possui como características principais:

1. Utiliza o tradutor de especificações algébricas, descrito no capítulo 3 para implementação de alguns de seus módulos.

2. Utiliza a implementação de semântica de ações existente no gerador de ações, produzido pelo sistema ABACO durante o processamento de especificações.

Esse capítulo esta organizado da seguinte forma:

- Na primeira parte desse capítulo, descreveremos algumas possíveis abordagens para implementação do processo de tradução e definiremos a abordagem que será utilizada durante esse capítulo.

- A seguir especificaremos o tradutor para semântica de ações que será divido em três partes:
  - A primeira parte apresentará a especificação de um simplificador/otimizador para semântica de ações.
  - A segunda parte descreverá o processo de produção do código para semântica de ações.
  - A terceira parte será responsável pela realização de otimizações no código produzido a partir do processo de tradução.
- Finalizando esse capítulo, descreveremos os resultados obtidos com uma implementação do tradutor da notação de ações descrito nesse capítulo.

6.1 Tradutor de Ações

O objetivo do tradutor da notação de ações, é produzir um código orientado a objetos, a partir de uma ação fornecida pelo usuário.

Na arquitetura do sistema ABACO descrito na Seção 5.5. A fase de tradução da notação de ações, corresponde as seguintes etapas:

- Simplificação da notação de ações: encarregada de processar a ação a ser traduzida realizando simplificações e otimizações de alto nível.
- Compilação de ações: encarregada de produzir o código para a ação resultante do processo de simplificação.
- Otimização de código: encarregada de realizar otimizações de baixo nível diretamente no código produzido pela etapa de geração de código.

essas etapas podem ser implementadas utilizando uma das seguites formas:

1. Através de uma especificação algébrica que será implementada pelo tradutor de especificações algébricas. Essa abordagem possui como vantagem a facilidade de criação e manipulação de estruturas de dados via especificações algébricas, além de gerar descrições mais elegantes.

2. Através de código orientado a objetos que utilize a biblioteca de classes produzida pela tradução da especificação de linguagens de programação. Essa abordagem possui como vantagem uma facilidade em expressar estados globais, tais como variáveis correntemente alocadas, e manipulação de entrada/saída.

Nesse capítulo, optamos por implementar as etapas do processo de tradução da notação de ações de uma forma mista, expressando a etapa de simplificação de ações através de especificações algébricas, aproveitando as facilidades para definição de transformações para a notação de ações, e as etapas de geração e otimização de código implementadas diretamente em C++, o que aproveita a facilidade de manipulação de estruturas de baixo nível.

As próximas seções descreverão cada uma dessas etapas.

6.2 Simplificador de Ações

Essa etapa tem o objetivo de processar a ação que deve ser traduzida, gerando uma representação mais eficiente e de fácil tradução. Para isso nós definiremos combinadores de ações mais abrangentes, que substituirão aqueles definidos pela notação de ações, e axiomas que poderão:

- Converter ações expressas através da notação de ações original em ações que utilizem a nova representação.

- Eliminar construções redundantes ou ineficientes.

O simplificador de ações que descreveremos nesse capítulo define uma extensão da notação de ações mostrada na Figura 6.1 Essa extensão da notação de ações define o com-

Notação de Ações Estendida

introduces: combinating _ _ _ _ _,
basically, sequencing, interleaving,
functionally, conducting, composing,
declaratively, overlaying, accumulating.

needs: Action Notation.

- combinating _ _ _ _ _ :: action, action, basically, functionally, declaratively $\rightarrow$ action.
- basically = sequencing **|** interleaving.
- functionally = conducting **|** composing.
- declaratively = conducting **|** overlaying **|**
  composing **|** accumulating.

Figura 6.1: Extensão da Notação de Ações

binador de ações: combinating x y b f d, onde:

- x e y são duas ações que serão combinadas.
- os operandos b f e d, definem a ordem de execução das sub-ações e forma com que as informações tempo'arias e de escopo serão processadas.

A forma com que o combinador processará as sub-ações é definida através dos seguintes sortes:

- basically: Define o comportamento básico da ação combinada. Os comportamentos básicos definidos são:
  - sequencing: A ação é basicamente sequencial. Isso significa que a primeira ação será executada e somente após o seu término é iniciada a execução da segunda ação.

  - interleaving: A ordem de execução das ações não é importante. As duas ações podem inclusive estarem executando intercaladamente.

- functionally: Define o comportamento funcional da ação combinada. O comportamento funcional de uma ação é definido pelos seguintes atributos:

  - conducting: Os dados transitórios recebidos pela ação combinada, são passados para as duas subações. Os transitórios produzidos pela ação combinada serão a união daqueles produzidos pelas subações.
  - composing: Os dados transitórios recebidos pela ação combinada são passados para primeira subação. A segunda subação será executada com as informações transitórias produzidas pela primeira subação e as informações transitórias produzidas por segunda subação formam o resultado da ação combinada.

- declaratively: Define o comportamento declarativo da ação combinada. O comportamento declarativo é determinado pelos seguintes valores:

  - conducting: De forma similar ao comportamento funcional, as associações recebidas pela ação combinada, são passadas para as duas subações. As associações produzidas pela ação combinada são formadas pela união disjunta das associações produzidas pelas subações.
  - composing: De forma similar ao comportamento funcional, As associações recebidas pela ação combinada são passadas para primeira subação. A segunda subação será executada com as associações produzidas pela primeira subação e as associações produzidas pela segunda subação formam o resultado da ação combinada.
  - overlaying: Similar ao comportamento conducting exceto que as associações produzidas pela ação combinada são formadas pelas associações produzidas pela primeira ação sobrepostas pelas associações produzidas pela segunda subação.
  - accomulating: A primeira subação recebe as associações recebidas pela ação combinada e a segunda subação recebe estas associação, acrescidas com as associações

needs: Action Notation,
Notação de Ações Estendida.

(1) a and b = combinating a b interleaving conducting conducting.

(2) a and then b = combinating a b sequencing conducting conducting.

(3) a then b = combinating a b sequencing composing conducting.

(4) a moreover b = combinating a b interleaving conducting overlaying.

(5) a hence b = combinating a b interleaving conducting composing.

(6) a before b = combinating a b sequencing conducting accumulating.

(7) furthermore x = rebind moreover x.

(8) fail or x = x.

(9) x or fail = x.

(10) escape trap x = x.

(11) x trap escape = x.

(12) (combinating x complete sequencing f d) = x.

(13) (combinating complete x sequencing f d) = x.

(14) (combinating fail x sequencing f d) = fail.

(15) (combinating escape x sequencing f d) = escape.

Figura 6.2: Especificação do Simplificador de Ações

produzidas pela primeira subação. O resultado da ação combinada é formado pela união das associações produzidas pelas duas subações.

O combinador combinating, será utilizado para expressar a semântica dos combinadores _ and _, _ and then _, _ then _, entre outros. Essa semântica, juntamente com algumas regras de otimização de ações, podem ser expressas através de especificações algébricas como podemos observar na Figura 6.2.

□

O uso de especificações algébricas para uma manipulação das especificações, possibilita uma descrição elegante das técnicas de otimização, além de permitir a inclusão de novas regras com relativa facilidade. No exemplo mostrado acima, somente colocamos as equivalências triviais dos combinadores de ações. Um possível estudo futuro seria a expansão desse exemplo de forma a inserir os algoritmos de análise e otimização da notação de ações propostos pelos sistemas geradores de compiladores descritos no Capítulo 5.

6.3 Descrição do Gerador de Código

Essa seção descreverá o processo de geração de código para a notação de ações estendida. Inicialmente, nós descreveremos as estruturas de dados que serão utilizadas pelo segmento de código gerado. A seguir descreveremos como será o comportamento do código produzido pela tradução de uma ação. Uma vez definidas essas características poderemos descrever os operadores responsáveis pela geração do código propriamente dito.

6.3.1 Estruturas de Dados

O código fonte produzido pelo gerador de código necessitará, para ser compilado, de algumas bibliotecas auxiliares que implementam tipos de dados utilizados em tempo de execução. Essas bibliotecas auxiliares serão estudadas nas próximas seções.

6.3.1.1 Biblioteca Auxiliar Dinâmica

A biblioteca auxiliar dinâmica da linguagem especificada, definida no Capítulo 5, contém a implementação dos sortes e operadores definidos pela notação de dados e pelas entidades semânticas da linguagem.

Essa biblioteca implementa os sortes e operadores segundo o tradutor de especificações algébricas descrito no Capítulo 3. O exemplo abaixo mostra como a biblioteca implementa os valores verdade da notação de ações.

Exemplo

O sorte truth-value, seus subsortes e operadores estarão definidos na biblioteca da seguinte forma:

- Todos os indivíduos do sorte truth-value são subclasses de `C_truth-value`.
- O sorte true é definido como um objeto da classe `Ci_true` de nome `Sort_true`.
- O sorte false é definido como um objeto da classe `Ci_false` de nome `Sort_false`.
- Operações como _ and _, _ or _,not _, etc, estarão definidas como métodos da classe `C_truth-value`.

dessa forma, expressões como:

(1) true or false.

são traduzidas para a seguinte expressão:

```
Sort_true->_or_(Sort_false).
```

□

O elementos definidos pela notação de dados serão traduzidos de forma similar ao sorte truth-value, visto acima. Os únicos sortes que não seguirão essa regra são os sortes numéricos e as strings, que serão representados através de classes especiais.

6.3.1.2 Biblioteca de Facetas

A biblioteca de facetas define um conjunto de estruturas de dados utilizadas para representar as informações produzidas e recebidas pela ação. Essa biblioteca é definida diretamente em C++ e não através de especificações algébricas por motivos de performance.

A biblioteca de facetas implementa as seguintes classes:

- `FailException`: Classe que representa a ocorrência de um término anormal da execução de uma ação. Essa classe possuirá uma única instância de nome `fail` que será utilizada durante a tradução do comando **fail**.

- `TrapException`: Classe que representa o desvio na execução de uma ação provocado pela ação **escape**. Essa classe armazena informação temporárias que serão passadas como para o combinador **trap**. São definidos os seguintes métodos para essa classe:

  - `TrapException(Transients t)`: Produz um objeto da classe `TrapException` que armazena as informações temporárias `t`.

  - `GetTransients()`: Obtem as informações temporárias armazenadas pela excessão.

- `Transients`: Representa as informações temporárias processadas pela ação. Essa classe armazena uma lista de objetos da classe `C_datum`, definida na biblioteca auxiliar dinâmica. A classe `Transients` possui os seguintes métodos:

  - `Transients()`: Gera um objeto da classe `Transients` que não armazena nenhuma informação.

  - `Transients(`d`)`: Produz um objeto da classe `Transients` formado pelo objeto da classe `C_data` que representa sortes da classe **data**.

  - t_1`->Union(`t_2`)`: Produz um objeto da classe `Transients`, formado pela união dos dados armazenados pelos objetos t_1, t_2 da classe `Transients`.

  - t`->Component(`n`)`: Essa expressão retorna o valor do n-ésimo componente existente na tupla t, se não existir nenhum componente é gerada uma exceção que represente o estado de término **failing**.

  - t`->Data()`: Produz um objeto da class `C_data` a partir do objeto t da classe `Transients`.

A classe `Transients` possui ainda um objeto de nome `empty_transients` que é usado para representar a ausência de informações transitórias.

- `Binds`: Representa as informações de escopo. Os objetos dessa classe armazenam listas de pares formados por inteiros e objetos da classe `C_bindable`, gerada pela tradução do sorte bindable. Durante o processo de tradução de ações, os tokens existentes na ação a ser traduzida são mapeados para números como forma de agilizar o processo de busca. A classe `Binds` define os seguintes métodos:

  - `Binds()`: Produz um binding vazio.

  - `Binds(`t`,`b`)`: Produz um objeto da classe `Binds`, que associa o identificador t com o valor b.

  - b_1`->Union(`b_2`)`: Produz um objeto da classe `Binds`, formado pela união das informações armazenadas nos objetos b_1 e b_2.

  - b_1`->Overlay(`b_2`)`: Produz um objeto da classe `Binds`, formado pelas informações armazenadas no objeto b_1, sobrepostas com as informações armazenadas no objeto b_2.

  - b`->At(`t`)`: Devolve o objeto da classe `C_bindable` associado ao identificador t.

  A classe `Binds` possui ainda um objeto de nome `empty_binds` que representa associações vazias.

- `C_internal_cell`: Representa uma célula de memória alocada. Essa classe, uma subclasse da classe produzida pela tradução do sorte cell, armazena um objeto storable e o estado de alocação da célula. Os objetos dessa classe possuem o seguinte comportamento:

  - c`->Store(`s`)`: Armazena o valor s na célula c.

  - c`->WhatStored()`: Recupera o valor armazenado na célula c.

  - c`->Deallocate()`: Marca a célula como estando desalocada. Após a chamada desse método, toda operação de acesso ao conteúdo dessa célula gerará um erro de execução.

- `Commited`: O comprometimento da execução de uma ação é representada através do flag `Commited` que inicialmente armazenará o valor `false`, sendo alterado pela execução da ação commit.

6.3.2 Características do Código Gerado

O resultado da tradução de uma ação programa será um comando válido da linguagem C++ com os seguintes pseudo-comandos adicionais:

- ```
  Unfolding(param-decl)
  {
    Bloco de comandos 1.
    unfold(params2)
    Bloco de comandos 2.  }(params1)
  ```

  Esse comando será utilizado para uma primeira implementação das ações unfold e unfolding a, por isso tem uma semântica semelhante à desses combinadores. A expressão `param-decl` contém declarações de variáveis que serão locais ao bloco de comandos. Inicialmente, o bloco de comandos será executado com os valores das varáveis locais iguais a `params1`. A cada execução de `unfold`, o bloco de comandos será executado novamente com os valores das varáveis locais igual a `params2`.

- `Abstraction(param-decl) {    Bloco de comandos. }`

  A expressão Abstraction retorna um objeto subclasse de `C_abstration`, que implementa o sorte abstraction. A sentença `param-decl` representa um conjunto de variáveis locais ao bloco de comandos que serão fornecidas como parâmetro. Esse objeto pode ser executado através da expressão `abst(params)`, que executará o bloco de comandos equivalente fornecendo `params` como parâmetros.

Esses pseudo-comandos serão tratados em uma fase posterior para a geração de um código C++ puro.
  ```

Durante a execução do código gerado pelo tradutor de especificações algébricas, os términos anormais das ações são representados pela ocorrência de uma exceções durante a execução do comando. O estado de término failing é representado pela exceção `FailException` e o estado escaping pela exceção `EscapeException`.

6.3.3 Funções de Tradução

Nessa seção apresentaremos o processo de tradução de ações e produtores.

O processo de tradução de uma ação é realizado pela função de tradução: $\mathsf{Tr}(a, t, b, t', b')$, onde:

- a é a ação a ser traduzida.
- t e b: identificam as variáveis que armazenam as informações temporárias e de escopo recebidos pela ação.
- t' e b': identificam as variáveis que armazenarão as informações temporárias e de escopo produzidos pela ação.

Essa função produz como resultado um comando que representa a tradução da ação fornecida.

Os produtores são traduzidos através da função de tradução: $\mathsf{Av}(p,t,b)$, onde:

- p é produtor que está sendo traduzido.
- t e b identificam os dados transitórios e bindings correntes.

O resultado dessa função é uma expressão C++ válida que representa a tradução do produtor.

Os tokens (strings) que representarão variáveis são traduzidos para números inteiros, através da função auxiliar $\mathsf{val}(x)$. Essa função retorna um inteiro que será associado à string x, essa função será necessariamente bijetiva.

As próximas seções definirão o comportamento das funções de tradução Tr e Av para ações primitivas, combinadas e para os produtores segundo a semântica natural da notação de ações descrita em [Mou93].

6.3.3.1 Ações Primitivas

O comando produzido pela aplicação do operador Tr(a,t,b,t',b'), para cada tipo de ação primitiva a é mostrado abaixo:

- A ação complete gera nenhum segmento de código e não produz nenhum tipo de informação. Portanto o resultado da tradução dessa ação será o comando: `skip`.
- A ação fail produz uma exceção que caracterizará a falha de execução e nenhum tipo de informação é produzido. Portanto o código gerado pela ação será:

  ```
  throw ExceptionFail;
  ```

- De forma similar à ação fail, a ação escape gerará uma exceção que transferirá a execução para a próxima ocorrência da ação trap. O código produzido será:

  ```
  throw ExceptionEscape(empty_transients);
  ```

- O código produzido para a ação escape with y, será:

  ```
  throw ExceptionEscape(Av(y, t, b));
  ```

- A ação commit gera o seguinte segmento de código:

  ```
  Commited = true;
  ```

- A ação rebind possui como efeito propagar as associações recebidas pela ação. A sua tradução produz o seguinte código:

  ```
  t' = empty_transients;
  b' = b;
  ```

- A ação give y fornece como informação transitória o valor produzido pela avaliação do produtor y, gerando o seguinte código:

  ```
  t' = Transients(Av(y,t,b));
  b' = empty_binds;
  ```

- A ação **bind** to **to** y produz uma associação entre o token to e o valor produzido pela avaliação do produtor y. O código gerado será o seguinte:

```
t' = empty_transients;
b' = Binds(val(to), Av(y,t,b));
```

- A tradução da ação **store** y_1 **in** y_2 utilizará o método `Store` das células para armazenar na célula produzida pela avaliação do produtor y_2, o valor produzido pela avaliação da célula y_1. O código resultante será o seguinte:

```
(Av(y2,t,b))->The_IndividualCell()->Store(Av(y1,t,b));
t' = empty_transients;
b' = empty_binds;
```

- A ação **allocate a cell** tem como efeito produzir uma nova célula de memória, que será fornecida como informação transitória. As células de memória, são traduzidas como objetos da classe `C_internal_cell`. Esse comportamento é traduzido no seguinte segmento de código:

```
t' = new C_internal_cell;
b' = empty_binds;
```

- A ação **deallocate** y tem como efeito liberar a célula de memória produzida pela avaliação do produtor y, não produzindo nenhum tipo de informação. O código resultante da tradução dessa ação será o seguinte:

```
Av(y,t,b)->Deallocate();
t' = empty_transients;
b' = empty_binds;
```

- O código produzido pela ação **enact** y executará a abstração produzida pela avaliação do produtor y. Isso é feito pelo seguinte código:

```
Av(y,t,b)(t,b,t',b');
```

- A ação **check** y verifica se o produtor y ao ser avaliado produz o valor **true**, falhando caso contrário. Isso é feito através do seguinte código:

```
if (Av(y,t,b)!=Sort_true)
   throw FailException;
t' = empty_transients;
b' = empty_binds;
```

- A ação **unfold**, será traduzida pelo comando:

```
unfold(t,b,t',b');
```

6.3.3.2 Combinadores de Ações

O comando resultante da aplicação do Tr(a,t,b,t',b') com ações combinadas é descrito abaixo:

- A ação "a_1 or a_2" produz o seguinte comando:

```
try
  {
   Tr(x,t,b,t',b')
  }
catch (FailException e)
  {
   if (!Commited)
    Tr(y,t,b,t',b');
   else
    throw;
  }
```

  onde x e y podem assumir os valores das ações a_1 a_2 em quaisquer ordem. O significado da tradução da ação é o seguinte: O código produzido pela tradução da primeira ação é executado, se durante a execução dessa ação ocorrer uma exceção do tipo FailException, o controle do programa será transferido para o bloco `catch`, que executará o código resultante da tradução da segunda ação. Alternativamente, a função de tradução poderia ainda decidir de forma não determinista qual das duas subações seria a primeira ação a ser executada.

- A ação "a_1 trap a_2" produz o seguinte comando:

```
try
  {
    Tr(a1,t,b,t',b')
  }
catch (TrapException e)
  {
    Transients t = e->GetTransients();
    Tr(a2,t,b,t',b');
  }
```

O significado da tradução da ação é o seguinte: O código produzido pela tradução da primeira ação é executado, se durante a execução dessa ação ocorrer uma exceção do tipo TrapException, o controle do programa será transferido para o bloco `catch`, que executará o código resultante da tradução da segunda ação.

- A ação "unfolding a" produz o seguinte comando:

```
Unfolding(C_data T1, C_bindings B1, C_data &T2,C_bindings &B2)
{
   Tr(a,T1,B1,T2,B2)
}(t,b,t',b');
```

A tradução do combinador unfolding simplesmente consiste em gerar um bloco utilizando o pseudo comando `Unfolding` que possui uma sintaxe semelhante à do combinador.

- A ação "combinating a_1 a_2 b f d" produz o seguinte comando:

```
{
   Tr(a1,t,b,tv,bv)
   Pt(f)
   Pb(d)
   Tr(a2,t'v,b'v,t',b');
   Rt(f)
   Rb(d)
}
```

onde:

- t_v, b_v,t'_v e b'_v: São variáveis locais temporárias que armazenam transitórios e bindings temporários.
- Pt : É uma função que tem como objetivo produzir os transitórios a serem recebidos pela ação a_2. O resultado da expressão Pt(f) é:
  - Se f= conducting, então o código produzido será:
    t'_v = t
  - Se f= composing, então o código produzido será:
    t'_v = t_v

– Pb : É uma função que tem como objetivo produzir os bindings a serem recebidos pela ação a_2. O resultado da expressão Pb(d) é:

  * Se d= conducting, então o código produzido será:
    b'_v = b
  * Se d= composing, então o código produzido será:
    b'_v = b_v
  * Se d= overlaying, então o código produzido será:
    b'_v = b
  * Se d= accumulating, então o código produzido será:
    b'_v = b_v`->overlay(`b`);`

– Rt : É uma função que tem como objetivo produzir os transitórios a serem produzidos pela ação combinada. O resultado da expressão Rt(f) é:

  * Se f= conducting, então o código produzido será:
    t' = t_v`->concat(`t'`);`
  * Se f= composing, então o código produzido será vazio.

– Rb : É uma função que tem como objetivo produzir os bindings a serem produzidos pela ação combinada. O resultado da expressão Pb(d) é:

  * Se d= conducting, então o código produzido será:
    b' = b_v`->Union(`b'`);`
  * Se d= composing, então o código produzido será vazio.
  * Se d= overlaying, então o código produzido será:
    b' = b'`->Overlay(`b_v`);`
  * Se d= accumulating, então o código produzido será vazio.

Observação: O processo de tradução descrito aqui para o combinador combinating traduz combinadores intelieving como se fossem sequenciais.

6.3.3.3 Produtores

Nessa seção apresentaremos o processo de tradução dos fornecedores e dos operadores da notação de dados. Essa tradução será representada operador: Av(y, t, b), onde:

- y representa o produtor a ser traduzido.
- t e b representam variáveis que armazenam os transitórios e bindings recebidos pelo produtor.

O resultado desse operador será uma expressão C++. A seguir vemos o resultado do operador de avaliação para cada tipo de produtor:

- O produtor **given** s verifica se o transiente corrente é do sorte s, caso positivo, retorna esse valor, a tradução desse produtor utiliza o método `Data` da classe `Transients` para recuperar valor armazenado. A verificação do sorte, para todos os produtores é feita através função `The_`s cuja semântica é produzir uma exceção se o objeto não for um subsorte de s. O código gerado será então o seguinte:

  t`->Data()->The_`s`()`

- A tradução do produtor **given** $s\#n$ utiliza o método `Component` para recuperar o n-ésimo valor armazenado nas informações transitórias. O código gerado será:

  t`->Component(`n`)->The_`s`()`

- A tradução do produtor **the** s **bound to** t utiliza o método `At` para recuperar o dado associado ao token val(t), gerando o seguinte código:

  b`->At(val(`t`))->The_`s`()`

- O produtor **the** s **stored in** y, utiliza o método `WhatStored` para recuperar o valor armazenado na célula produzida pela avaliação de y, isso gera o seguinte código:

  `Av(`y`,`t`,`b`)->The_IndividualCell()->WhatStored()->The_`s`()`

- O produtor formado pela expressão : o(s_1,s_2,...), é traduzido de acordo com o especificado pelo processo de tradução de especificações algébricas visto no Capítulo 3, ou seja, o objeto gerado pela tradução do produtor s_1 deve responder a uma mensagem de nome o, implementada na biblioteca auxiliar dinâmica. Por isso a tradução gera o seguinte código:

  `(avaliacao(`s_1`,`t`,`b`))->o((avaliacao(`s_2`,`t`,`b`)),...);`

- Os nomes de sortes definidos na notação de dados ou nas entidades semânticas, são traduzidos como acessos à variavel produzida pelo processo de tradução para representar esse sorte.

- O produtor formado pelo número n produz um objeto da classe que representa os números inteiros, gerando o seguinte código:

  ```
  new InternalNumber(n)
  ```

- O produtor formado pela string s, produz um objeto da classe que representa as string, gerando o seguinte código:

  ```
  new InternalNumber(s)
  ```

- O produtor abstration of a produz um objeto da classe C_abstraction que recebe como parametros variáveis que armazenam informações temporárias e de escopo recebidas e produzidas pela ação durante sua execução, o código a ser executado é formado pela tradução da ação a. O código resultante da tradução desse produtor será:

  ```
  Abstraction(C_data T1, C_bindings B1, C_data &T2, C_bindings &B2)
   {
     Tr(a,T1,B1,T2,B2)
   }
  ```

- O produtor closure abstration of a gera um código semelhante ao mostrado no ítem anterior, a única diferença é que a ação a recebe o escopo existente durante a avaliação do produtor e não o escopo existente durante a execução da ação. O código gerado é mostrado abaixo:

  ```
  Abstraction(C_data T1, C_bindings B1, C_data &T2, C_bindings &B2)
   {
     Tr(a,T1,b,T2,B2)
   }
  ```

6.4 Otimizador do Código Gerado

O código produzido pelo gerador de código deve ainda passar por uma fase de otimização que realizará, entre outras coisas:

- Uma otimização do uso de variáveis e constantes;
- Uma substituição de objetos por valores nativos da linguagem.

6.4.1 Otimização de Variáveis e Constantes

Esse tipo de otimização procura melhorar o código gerado pela notação de ações eliminando sentenças como as mostradas abaixo:

- Sentenças como:

  ```
  b = BIND(1,x);
  ....
  expr(...,b->At(1),...);
  ```

  podem ser substituídas por:

  ```
  v = x;
  b = BIND(1,v);
  ....
  expr(...,v,...);
  ```

  onde v é uma nova variável.

- Sentenças como:

  ```
  t = TRANSIENTS(x);
  ....
  expr(...,t->Component(1),...);
  ```

  podem ser subsituidas por:

  ```
  v = x; t = TRANSIENTS(v);
  ....
  expr(...,v,...);
  ```

  onde v é uma nova variável.

- Atribuições que nunca forem realmente utilizadas podem ser eliminadas.
- Expressões como:

  ```
  exp->The_x()
  ```

se exp é uma expressão cujo tipo é uma subclasse da classe genérica gerada pelo sorte x então a chamada ao método `The_`x pode ser eliminada.

6.4.2 Substituição de Objetos por Valores

O objetivo desse tipo de otimização é a eliminação de operações de criação de objetos e chamadas a métodos que manipulam tipos pré-definidos de C++ (inteiros e booleanos), substituindo-as pelos operadores da linguagem. Esse tipo de otimização é crucial para a obtenção de compilador de boa qualidade, uma vez que o principal overhead da programação orientada a objetos é o custo da criação de objetos e de chamadas a métodos.

A seguir citaremos alguns exemplos desse tipo de otimização:

1. Segmentos de código:

   ```
   x = y->sum_(z);
   ```

   onde y e z são expressões que retorna objetos da classe `Integer`. Podem ser substituídos pela seguinte versão mais eficiente:

   ```
   x = new Integer(y->val+z->val);
   ```

   onde `val` é uma variável da classe `Integer` que armazena o valor do inteiro representado. As mesmas otimizações podem ser feitas para as operações: difference, product, division, etc.

2. Expressões como:

   ```
   new Integer(x)->val
   ```

   podem ser substituídas por:

   ```
   x
   ```

3. Expressões e comandos de manipulação da memória como:

   ```
   x->Store(y)
   ```

x`->WhatStored()`

podem ser substituídas, respectivamente, por:

x`->value =` y`;`

x`->value`

onde `value` é a variável da classe `InternalCell` responsável por registrar o valor armazenado na célula.

O uso de otimizações como as acima, melhora sensivelmente o resultado da tradução, entretanto, elas diminuem a flexibilidade do gerador de código pois fixas as semânticas das operações otimizadas impedindo que novas características sejam acrescentadas a esses operadores. Um exemplo dessa limitação seria a da operação `sum`, que define a some de números, se esta for expandida para significar também a soma de strings esse tipo de otimização não pode mais ser aplicado.

6.5 Geração do Código Final

A última parte do processo de geração de código para a notação de ações consiste em produzir o código final que poderá ser compilado. Para isso o comando otimizado gerado pelos passos anteriores passa pelas seguintes transformações que eliminarão os comandos `abstraction` e `unfolding`:

- Os pseudo comando `unfolding` pode ser tratado de duas formas diferentes:
  1. Se dentro do bloco `unfolding` os comandos `unfold` estiverem localizados no fim de cada bloco (recursão de cauda), o comando `unfolding` pode ser substituído por um bloco de comandos C++ e as chamadas `unfold` por comandos `goto`.
  2. Se o bloco não puder ser enquadrado no item acima, o comando `unfolding` é transformado em uma nova função C++, as variáveis utilizadas dentro do bloco `unfolding` são transformadas em variáveis locais a esse novo método e

a ocorrência do comando `Unfolding` dentro do comando é substituída por uma chamada recursiva a esse método.

- Segmentos que usem a pseudo expressão `abstraction` { `command` } como:

```
main()
{
  ...
  var = abstraction(decl) { comando };
  ...
  var(params);
  ...
}
```

são traduzidos para código C++ puro da seguinte forma:

```
void fun(decl)
{
  comando
}
main()
{
  ...
  var = new C_abstraction(fun);
  ...
  var->Run(params);
  ...
}
```

6.5.1 Exemplo

Como exemplo do processo descrito nesse capítulo, mostraremos o resultado da tradução de um pequeno programa escrito na linguagem especificada na Seção 4.4. O programa que iremos traduzir é mostrado abaixo.

```
  var x := 0
in
  x := x + 1
```

```
end
```

Esse programa foi processado por um analisador sintático que produziu uma árvore sintática abstrata que foi submetida às funções semânticas da especificação, gerando a seguinte ação:

```
| | | give 0
| | and
| | | allocate-a-cell
| then
| | | bind "x" to the given cell # 2
| | and
| | | store the given value # 1 in the given cell # 2
hence
| | | | give the storable storedin the cell boundto "x"
| | | and
| | | | give 1
| | then
| | | give (sum the given integer # 1) (the given integer # 2)
| then
| | store the given value in the cell boundto "x"
```

A ação produzida, submetida às regras de simplificação mostradas na Seção 6.2, será convertida na seguinte expressão antes de ser traduzida:

```
Combinating sequencing conducting composing
    Combinating sequencing composing conducting
        Combinating interleaving conducting conducting
            give(0)
            allocate-a-cell
        Combinating interleaving conducting conducting
            Combinating interleaving conducting conducting
                bind("x")to((the given cell # 2))
                (store (the given value # 1) in (the given cell # 2))
    Combinating sequencing composing conducting
        Combinating sequencing composing conducting
            Combinating interleaving conducting conducting
                give(the(storable)storedin(the(cell)boundto("x")))
                give(1)
            (give (sum (the given integer # 1) (the given integer # 2)))
        store((the given value))in(the(cell)boundto("x"))
```

Essa expressão é traduzida utilizando as regras mostradas na Seção 6.3, produzindo o seguinte trecho de código:

```
Transients t0;
Transients t1;
Transients t2;
Binds b0;

t0 = Transients(new InternalInteger(0));
t1 = Transients(new IndividualCell);
t0 = t0.Concat(t1);
b0 = Binds(0, t0.Find(2)->The_cell());
t0.Find(2)->The_cell()->Store(t0->Find(1)->The_storable());
t1 = Transients(b0.At(0)->The_cell()->WhatStored());
t2 = Transients(new InternalInteger(1));
t1 = t1.Concat(t2);
t2 = Transients(t1.Find(1)->The_integer()->sum__(t1.Find(2)->The_integer()));
b0->At(0)->The_cell()->Store(t2->The_storable());
```

A fase de otimização de código de baixo nível processará o fonte acima gerando o seguinte código final:

```
{
```

```
  Transients t0;
  Transients t1;
  Transients t2;
  Binds b0;
  C_cell *cell0;

  t0 = TRransients();
  cell0 = new IndividualCell;
  b0 = Binds(0, cell0);

  cell0->value = new InternalInteger(0);

  t2 = cell0->value->The_integer()->val + 1;
  cell0->value = new Integer(cell0->value->The_integer()->val + 1);

}
```

6.6 Eficiência da Tradução

Para ter uma idéia da performance de execução do código traduzido, fizemos um teste com a tradução do programa mostrado na Figura 6.3, escrito na linguagem SIMPLE especificada na Seção 4.4.

A especificação da linguagem foi processada pelo sistema ABACO, descrito no Capítulo 5, que utilizou uma implementação parcial do método proposto nesse capítulo (que não inclui a etapa de otimização do código fonte). O compilador gerador foi capaz de processar o teste de tradução, gerando um programa executável.

Nesse teste procuramos verificar os seguintes itens:

- Qual o overhead gerado pelo programa por usar orientação a objetos na representação dos tipos básicos.
- Qual o overhead gerado pelo método de tradução

Para isso testamos o programa acima nas seguintes situações:

- Compilamos o programa equivalente em C++ utilizando um compilador escrito manualmente (GNU CC).

```
var m := 0;
var n := 100000;
var s := 0;
in
  while ( n > 0 ) do
    s = s + n;
    n = n - 1;
  end
```

Figura 6.3: Teste de Tradução

- Compilamos o programa de acordo com o método descrito nesse capítulo, utilizando a implementação sem nenhuma otimização.
- Compilamos o programa, otimizado manualmente de acordo com as regras existentes na Seção 6.4. Para termos uma idéia do ganho de cada tipo de otimização, testamos o efeito de cada uma delas separadamente.

Os tempos de execução dos executáveis obtidos são mostrados na Tabela 6.1. Nessa figura podemos tirar as seguintes conclusões:

1. O processo de geração de código utilizando orientação a objetos é bastante ineficiente, no nosso exemplo o código executou cerca de 800 vezes mais lentamente do que o executável gerado pelo GNU CC.
2. A utilização de técnicas para reduzir o uso de objetos pelo código proporcionou um aumento de 30 vezes na sua velocidade de execução, o que torna evidente a importância dessas regras para a performance do compilador. Novas técnicas podem ainda ser propostas para uma maior eficiência do código.

Situação	**Tempo de Execução**
Compilador GNU CC	0,2
Código automático	160,3
Código automático utilizando otimizações de variáveis	103,5
Código automático utilizando todas as otimizações	3,9

Tabela 6.1: Testes de Performance (em segundos).

Capítulo 7

Conclusões

Nesse capítulo descreveremos as principais contribuições apresentadas por esta dissertação. Na primeira seção comentaremos os resultados obtidos com as técnicas propostas. A seguir proporemos alguns tópicos de pesquisa que darão continuidade a este trabalho e, finalmente, mostraremos as principais conclusões que puderam ser obtidas com o uso do sistema Abaco.

7.1 Resultados Obtidos

Propomos um sistema gerador de compiladores, denominado Abaco, cuja principal característica é ser baseado na descrição algébrica da notação de ações permitindo, entre outras coisas, que:

- O sistema possa ser facilmente expandido, através da adição da especificação de novas estruturas ao núcleo do mesmo.
- O compilador gerado, por utilizar bibliotecas geradas a partir da especificação da linguagem de programação em lugar de bibliotecas estáticas, reflete melhor as particularidades da linguagem especificada. A principal vantagem dessa característica é a possibilidade de suporte a linguagens que possuam propriedades não suportadas pelas entidades semânticas tais como classes e objetos.

Propomos também nesta dissertação um método de tradução de especificações algébricas unificadas, que mostrou ser útil na prototipação de especificações algébricas genéricas, tendo sido utilizado ainda como base do sistema ABACO.

Para demonstrarmos a viabilidade desses sistemas construímos uma implementação dos mesmos. Essa implementação foi escrita em C++ e possui aproximadamente 10000 linhas de código não gerado automaticamente que estão divididos nas seguintes partes:

- Uma implementação do método de tradução de especificações algébricas, responsável pelo processamento das especificações de linguagens de programação. Esse programa é formado pelos seguintes módulos:
  1. Um parser de especificações algébricas (900 linhas).
  2. O algoritmo de tradução dessas especificações. (5000 linhas)
  3. Um módulo responsável adicionar segmentos de código no código gerado responsáveis pela desalocação de memória. (1800 linhas)
  4. Bibliotecas auxiliares (800 linhas)
- Uma implementação do método de tradução de ações, formado por:
  1. O algortimo de tradução de ações (1100 linhas)
  2. A biblioteca de facetas (200 linhas).

Além do código produzido em C++, foi definida uma especificação da notação de ações e de dados (mostrada nos apêndices B C), essas especificações foram processadas pelo tradutor de especificações algébricas segundo o especificado pelo sistema ABACO, produzido automaticamente um biblioteca de cerca de 20000 linhas de código.

7.2 Trabalhos Futuros

Os seguintes tópicos de pesquisa podem vir a ser estudados como continuação dos trabalhos dessa dissertação:

1. Na área de prototipação de especificações algébricas:

   - Uma verificação da corretude do método de tradução de especificação algébricas, com o objetivo de garantir que o código gerado por uma especificação possua realmente o comportamento especificado.
   - Após a definição das regras de tradução, foi feita apenas um estudo superficial de técnicas de otimização do código gerado. Um estudo aprofundado de novas regras de otimização contribuiria para uma melhora no método.
   - A boa performance do código gerado pelo método de tradução de especificação algébricas unificadas sugere estudos sobre a aplicação dessa técnica a outros formalismos de especificações algébricas como OBJ [GKK+88],
   - O tradutor de especificações algébricas insere no código gerado algumas rotinas responsáveis pela desalocação dos objetos. Essas rotinas não foram trabalhadas para se conseguir um maior desempenho no código gerado, por esse motivo comparações de performance mostraram uma penalidade de cerca de 30% devido ao uso dessas rotinas. A inserção de técnicas otimizadas para desalocação de memória tornaria os programas gerados sensivelmente mais eficientes.

2. Na área de geração automática de compiladores:

   - A inserção de novas técnicas de análise e otimização de ações, descritas em sistemas geradores de compiladores como os mostrados no Capítulo 5, no núcleo do sistema Abaco, pode melhorar a performance dos compiladores gerados.
   - A definição de novas técnicas de análise que permitam uma melhor utilização dos tipos de dados primitivos de C++ em substituição às chamadas de métodos.
   - O teste do sistema com exemplos complexos como a especificação semântica de linguagens orientadas a objetos, que utilizam novas estruturas de dados diferentes das definidas pela notação de ações.

7.3 Considerações Finais

Os estudos realizados nessa dissertação produziram:

- Um estudo de equivalências entre especificações algébricas e linguagens orientadas a objetos. Essas equivalências possibilitaram a criação de um método automático de tradução que apresentou resultados satisfatórios.

- Um método para geração de compiladores orientados a objeto. Esses compiladores gerados possuem como principal característica a facilidade de adição de novas propriedades ao sistema. Um grande problema apresentado com os compiladores gerados é a baixa performance dos códigos compilados.

Referências Bibliográficas

[ASU86] Alfred V. Aho, Rave Sethi, and Jeffrey D. Ullman. *Compilers - Principles Techiniques and Tools*. Addison Wesley Publishing Company, 1986.

[BMW92] D. F. Brown, H. Moura, and D. A. Watt. ACTRESS: an action semantics directed compiler generator. Departmental Research Report FM-1992-1, University of Glasgow, Department of Computing Science, June 1992.

[Bro97] Deryck F. Brown. *Sort Inference in Action Semantics*. PhD thesis, Department of Computing Science, University of Glasgow, 1997.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. *Rewrite Systems*, chapter 6. Elsevier Science Publishers, 1990.

[GKK+88] Joseph Goguen, Claude Kirchner, Hélèno Kirchner, Aristide Mégrelis, and José Meseguer. An introduction to obj3. In *Conference on Conditional Term Rewriting*, volume 308 of *Lecture Notes in Computer Science*, Springer 1988.

[GM92] Joseph Goguen and José Meseguer. Order-sorted algebra i: Equational deduction for multiple inheritance and partial operations. *Theoretical Computer Science*, 2, 1992.

[Gogar] Joseph A. Goguen. *Theorem Proving and Algebra*. To appear.

[Gue81] Irene Guessarian. Algebraic semantics. *Number 99 in Lecture Notes in Computer Science*, 1981.

[Gue95] Luiz Carlos Castro Guedes. *Um Modelo Orientado a Objetos para Geração Automática de Compiladores.* PhD thesis, Pontificia Universidade Católica - Rio Janeiro, 1995.

[Ham95] Lutz H. Hamel. *Behavioural Verification and Implementation of an Optimising Compiler for OBJ3.* PhD thesis, Oxford University Computing Laboratory, 1995.

[Joh79] S. C. Johnson. Yacc - yet another compiler compiler. Technical report, Bell Laboratories, 1979.

[Lam85] Leslie Lamport. *LaTeX A Document Preparation System.* Addison-Wesley Publishing Company, 1985.

[LS79] M. Lesk and E. Schmidt. Lex - a lexical analyzer generator. Technical report, Bell Laboratories, 1979.

[Mey97] Bertrand Meyer. *Object Oriented Software Construction.* Prentice Hall, 1997.

[Mos89] P. D. Mosses. Unified algebras and action semantics. In *Proceedings of STACS'89.* Springer-Verlag, 1989. Lecture Notes in Computer Science, volume 349.

[Mos92] Peter D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[Mou93] H. Moura. *Action Notation Transformations.* PhD thesis, University of Glasgow, Department of Computing Science, 1993.

[MW93] Peter D. Mosses and David A. Watt. Pascal action semantics, version 0.6. Available by FTP as ftp.brics.dk/pub/BRICS/Projects/AS/Papers/-MossesWatt93DRAFT/pas-0.6.ps.Z, March 1993.

[Ørb93] Peter Ørbæk. Analysis and optimization of actions. M.Sc. dissertation, Depto. of Computer Science, Univ. of Aarhus, September 1993.

[Pal92] Jens Palsberg. *Provably Correct Compiler Generation.* PhD thesis, Dept. of Computer Science, Univ. of Aarhus, 1992.

[Sch86] D. A. Schmidt. *Denotational Semantics.* Allyn & Bacon, 1986.

[SF90] George Springer and Daniel P. Friedman. *Scheme and the Art of Programming.* MIT Press and McGraw Hill, 1990.

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1991.

[Wat88] David A. Watt. An action semantics of Standard ML. In *Proc. Third Workshop on Math. Foundations of Programming Language Semantics, New Orleans*, volume 298 of *Lecture Notes in Computer Science*, pages 572–598. Springer-Verlag, 1988.

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

[WS91] Larry Wall and Randal L. Schwartz. *Programming in Perl.* O'Reilly and Associates, 1991.

Apêndice A

Exemplo de Tradução

Esse apêndice fornece uma descrição detalhada do exemplo de tradução de especificações algébricas mostrado resumidamente no Capítulo 3

A.1 Especificação

Números Naturais

introduces: natural, positive-integer, sucessor _, sum _ _,0, fib _.

(1) sucessor _ :: natural $\rightarrow$ natural ($total$, $injective$).

(2) 0 : natural.

(3) sum 0 $n = n$.

(4) sum sucessor m n = sucessor sum m n.

(5) fib _ :: natural $\rightarrow$ natural.

(6) fib 0 = 0.

(7) fib sucessor 0 = sucessor 0.

(8) fib sucessor sucessor x = sum (fib sucessor x) (fib x).

Pilhas

introduces: stack, empty-stack, item, push _ _, pop _, top _.

needs: Numeros Naturais.

(1) push _ _ :: item , stack $\rightarrow$ stack.

(2) pop _ :: stack $\rightarrow$ stack.

(3) top _ :: stack $\rightarrow$ item.

(4) item $\geq$ natural.

(5) empty-stack : stack.

(6) top push n p = n.

(7) pop push n p = p.

(8) top empty-stack = nothing.

(9) pop empty-stack = nothing.

A especificação que desejamos traduzir é a de pilhas, essa especificação necessita da definição dos números naturais cuja especificação também é mostrada.

A.2 Modelo Simplificado

O primeiro passo para a obtenção da Especificação Algébrica Unificada Simplificada é eliminarmos as sentenças **needs** da especificação. Essa sentença é removida da especificação de pilhas adicionando-se à especificação a assinatura e os axiomas existentes na especificação dos números naturais. Nas duas especificações a maioria das declarações de funcionalidades e dos axiomas estão de acordo com as exigências do modelo simplificado, a única exceção é o axioma:

item $\geq$ natural.

da linha 4 da especificação de pilhas. Para simplificarmos a especificação, esse axioma é substituído por sua forma simplificada:

item = item | natural.

dessa forma a especificação simplificada resultante será:

introduces: natural, positive-integer, sucessor _, sum _ _,0, fib _,
stack, empty-stack, item, push _ _, pop _, top _.

(1) sucessor _ :: natural $\rightarrow$ natural (*total, injective*).

(2) 0 : natural.

(3) sum 0 $n = n$.

(4) sum sucessor m n = sucessor sum m n.

(5) fib _ :: natural $\rightarrow$ natural.

(6) fib 0 = 0.

(7) fib sucessor 0 = sucessor 0.

(8) fib sucessor sucessor x = sum (fib sucessor x) (fib x).

(9) push _ _ :: item , stack $\rightarrow$ stack.

(10) pop _ :: stack $\rightarrow$ stack.

(11) top _ :: stack $\rightarrow$ item.

(12) item = item | natural.

(13) empty-stack : stack.

(14) top push n $p = n$.

(15) pop push n $p = p$.

(16) top empty-stack = nothing.

(17) pop empty-stack = nothing.

Termos Iguais	**SuperSortes**	**SubSortes**	**É-elemento**
natural	item	0, sucessor natural	não
0, fib 0	natural, item		sim
sucessor natural	natural, item	sucessor 0	não
sucessor 0, fib sucessor 0	sucessor natural, ...		sim
fib natural	natural, item		não
item, natural $\mid$ item		0, natural	não
stack		empty-stack,...	não
empty-stack, pop empty-stack ...,	stack		sim
top stack	stack		não
pop stack	stack		não
push item stack	stack		não
nothing	todos		não

Tabela A.1: Sortes identificados na especificação de pilhas

A.3 Identificação dos Sortes

Essa etapa tem como objetivo identificar os principais termos que geram sortes na especificação. A análise da especificação simplificada gerou a seguinte listagem de termos:

- 0, natural, sucessor natural, sucessor 0, sum natural natural, fib natural, item, stack, item | stack, empty-stack, top stack, pop stack, push item stack, nothing.

O próximo passo é a identificação das relações existentes entre esses termos. O estudo dos axiomas da especificação origina a tabela de sortes mostrada na Tabela A.1

A.4 Geração do Esqueleto de Classes

As informações produzidas pela etapa de identificação de sortes definirão a hierarquia de classes mostrada na Figura A.1.

As classes de nome G(x) representam a classe genérica gerada pelo sorte x. Essas classes não definem nenhum objeto.

As classes de nome E(x) representam a classe específica gerada pelo sorte x, essa classe possuirá um objeto que representará o sorte x em expressões.

As classes na forma o (x_1,x_2,..,x_n) denotam classes operadoras que representam termos gerados pelo operador o, com parâmetros dos sortes x_1,x_2,...x_n, essas classes possuirão variáveis de instância para representar cada parâmetro e um método para inicializar essas variáveis. Por exemplo a classe definida pela expressão sucessor natural é a seguinte:

```
class Co_sucessor_natural
{
  G(natural) v1;
  C_natural Init(G(natural) p0)
  {
    v1 = p0;
    return this;
  }
};
```

Observações: A seguir faremos alguns comentários sobre a Figura A.1:

- Na figura, o termo sucessor 0 irá gerar uma classe que não é necessária na especificação. Esse é um efeito colateral do suporte a especificações da forma:

  bindings = map [token to bindable].

  op _ _ ... :: bindings, ...

  cuja característica é que um termo operador (map [token to bindable]), produz um novo conjunto de entidades que possuirão um comportamento distinto dos outros sortes formados pelo operador (map [...], no exemplo).

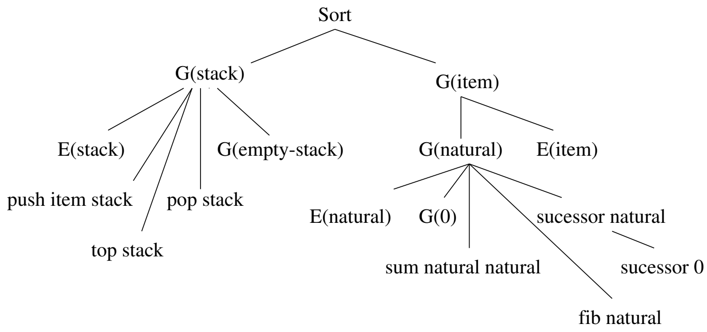

Figura A.1: Hierarquia Simplificada de Classes Gerada pela Especificação de Pilhas

- As classes operadoras: pop stack, fib natural, top stack. Não são utilizadas pelo código fonte pois todos os termos gerados pelas operação pop,top e fib, são reduzidas pelas equações de reescrita. A sua geração não pode ser evitada devido a falta de um processo de análise que pudesse verificar na necessidade de geração dessas classes.

A.5 Geração dos Métodos Auxiliares

Os principais métodos auxiliares produzidos pelo tradutor são:

A.5.1 Métodos The_x

Os métodos da forma The_x são gerados por cada sorte definido pela especificação. O sorte natural, por exemplo, define um método de nome The_natural, com os seguintes comportamentos:

- Para a classe Sort é definido o seguinte método:

```
C_natural The_natural()
{
  return NULL;
```

```
}
```

- Para a classe `G(natural)` é definido o seguinte método:

```
C_natural The_natural()
{
  return this;
}
```

A.5.2 Métodos `Match_`x

Os métodos da forma `Match_`x são definidos por cada operador existente na especificação. Por exemplo, o operador `sucessor _` define um método de nome `Match_sucessor_(G(natural) & param0)`. Esse método possui as seguintes versões:

- Para a classe `Sort` é definido o seguinte método:

```
Match_sucessor_(G(natural) & param0)
{
  return false;
}
```

- Para a classe `G(natural)` é definido o seguinte método:

```
bool Match_sucessor_(G(natural) & param0)
{
   param0 = v0;
   return true;
}

```

A.6 Geração dos Métodos Operadores

As próximas seções mostrarão o código produzido pela tradução dos operadores.

Operador sucessor _

O operador sucessor_ gera um método de nome sucessor_ para a classe G(natural) com o seguinte código:

```
1  G(natural) sucessor_()
2  {
3    if ((this)->IS(Sort_0))
4      return Sort_sucessor_0;
5    if ((this)->ISSuperSorte(Sort_natural))
6      return new Co_sucessor_natural ( (this)->The_natural() ) ;
7  }
```

Como podemos observar, o código não gera nenhum código para verificação de axiomas (dado a inexistência de axiomas). O objetivo da primeira parte do código (linhas 3-4) é verificar se o termo corresponde à constante determinada. O código restante (linhas 5-6) geram objetos da classe operadora.

Operador sum _ _

O operador sum _ _ gera um método de nome sum__ para a classe G(natural) com o seguinte código:

```
1  G(natural) sum__(G(natural) *p1)
2  {
3    {
4      G(natural) *  local0 = NULL;
5
6      if ((p1)->Match_sucessor_(local0))
7        return ((this)->sum__(local0))->sucessor_();
8    }
9    if (p1==Sort_0)
10     return this;
11   if (((this)->ISSuperSorte(Sort_natural)&&(p1)->ISSuperSorte(Sort_natural)))
12     return new Co_sum_natural_natural ( (this)->The_natural(), (p1)->The_natural() ) ;
13 }
```

As linhas (3-8) são geradas a partir do axioma:

sum x sucessor y = sucessor sum x y.

As linhas (9-10) são geradas pelo axioma:

sum x 0 = x.

As linhas 11-12 tem como objetivo produzir objetos da classe operadora correspondente, se nenhuma equação de reescrita puder ser aplicada. Esse código nessa especificação nunca será executado, uma vez que todos os termos da forma (sum x y) serão reduzidos pelos axiomas da especificação.

Operador fib _

O operador fib _, produzirá um método de nome `fib__` para a classe G(natural) com o seguinte código:

```
G(natural) fib_()
{
   if (this==Sort_0)
     return Sort_0;

  {
    G(natural) *  local0 = NULL;

    if ((p1)->Match_sucessor_(local0))
      if (local0==Sort_0)
         return Sort_0->sucessor_();
  }

  {
    G(natural) *  local0 = NULL;

    if ((this)->Match_sucessor_(local0))
      {
        G(natural) *  local1 = NULL;

        if ((local0)->Match_sucessor_(local1))
          return (((local1)->sucessor_())->fib_())->sum__((local1)->fib_());
      }
  }

  if ((this)->ISSuperSorte(Sort_natural))
```

```
27     return new Co_fib_natural ( (this)->The_natural() ) ;
28
29 }
```

As linhas (3-4) são geradas pelo axioma:

fib 0 = 0.

As linhas 6-12 são geradas pelo axioma:

fib sucessor 0 = sucessor 0.

As linhas 14-24 são geradas pelo axioma:

fib sucessor sucessor x = sum (fib x) (fib sucessor x).

Finalmente as linhas a partir da 25 tem como objetivo produzir objetos da classe operador e, de forma análoga ao operador sum _ _, nunca será executada.

Operador push _ _

O operador push _ _, produzirá um método de nome push__ para a classe G(item) com o seguinte código:

```
1 G(stack) push__(G(stack) *p1)
2 {
3   if (((this)->ISSuperSorte(Sort_item)&&(p1)->ISSuperSorte(Sort_stack)))
4     return new Co_push_item_stack ( (this)->The_item() , (p1)->The_stack() ) ;
5 }
6
```

As linhas 3-4 geram objetos da classe operadora correspondente. No caso desse operador, não existe nenhum axioma que possa reduzir termos da forma push x y.

Operador top _

O operador top _ _, produzirá um método de nome `top__` para a classe G(item) com o seguinte código:

```
1  G(item) top_()
2  {
3    if (this==Sort_empty−stack)
4       return Sort_nothing;
5    {
6      G(item) * local0 = NULL;
7      G(stack) * local1 = NULL;
8
9      if ((this)->Match_push__(local0 , local1))
10       return local0;
11   }
12   if ((this)->ISSuperSorte(Sort_stack))
13     return new Co_top_stack ( (this)->The_stack() ) ;
14 }
15
```

As linhas 3-4 são geradas pelo axioma:

top empty-stack = nothing.

As linhas 5-11 são geradas pelo axioma:

top push x y = x.

As linhas (12-13) produzem objetos da classe operador e nunca será executada pelo mesmo motivo do operador sum _ _.

Operador pop _

O operador pop _ _, produzirá um método de nome `pop__` para a classe G(item) com o seguinte código:

```
G(stack) pop_()
{
```

```
3   if (this==Sort_empty-stack)
4      return Sort_nothing;
5   {
6     G(item) *  local0 = NULL;
7     G(stack) *  local1 = NULL;
8
9     if ((this)->Match_push__(local0 , local1))
10      return local1;
11  }
12  if ((this)->ISSuperSorte(Sort_stack))
13    return new Co_pop_stack ( (this)->The_stack() ) ;
14 }
15
```

As linhas 3-4 são geradas pelo axioma:

pop empty-stack = nothing.

As linhas 5-11 são geradas pelo axioma:

pop push x y = y.

As linhas 12-13 produzem objetos da classe operador e nunca será executada pelo mesmo motivo do operador sum _ _.

A.7 Biblioteca Gerada

A seguir mostramos o código fonte formado pela a união dos segmentos de código descritos nas seções anteriores que se constitui no código final da biblioteca. O código que mostramos aqui consitui uma simplificação, sem colocarmos todos os métodos repetitivos como os `The_`x, `Match_`x, etc.

```
class Sort :  virtual public  Object
{

  bool Match_push__(C_item & param0,C_stack & param1)
  {
    return false;
```

```
  }

  *********************************
  ******* Outros Metodos Match ******
  *********************************

  C_stack The_stack()
  {
    return NULL;
  }

  *********************************
  ******* Outros Metodos The ********
  *********************************

};

class C_stack : virtual public Sort
{

  C_stack pop_()
  {
    {
      C_item local0 = NULL;
      C_stack local1 = NULL;

      if ((this)->Match_push__(local0 , local1))
        return local1;
    }
    if ((this)->ISSuperSorte(Sort_stack))
      return new Co_pop_stack( (this)->The_stack() ) ;
  }

  C_item top_()
  {
    {
      C_item local0 = NULL;
      C_stack local1 = NULL;

      if ((this)->Match_push__(local0 , local1))
        return local0;
    }
    if ((this)->ISSuperSorte(Sort_stack))
      return new Co_top_stack( (this)->The_stack() ) ;
  }

  C_stack The_stack()
  {
    return this;
  }
```

```
};

class Ci_stack : virtual public C_stack
{

};
C_stack Sort_stack = new Ci_stack;

class C_item : virtual public Sort
{

  C_stack push__(C_stack p1)
  {
    if (((this)->ISSuperSorte(Sort_item)&&(p1)->ISSuperSorte(Sort_stack)))
      return new Co_push_item_stack( (this)->The_item() , (p1)->The_stack() ) ;
  }

};

class Ci_item : virtual public C_item
{

};
C_item Sort_item = new Ci_item;


class C_empty-stack : virtual public C_gen-s4
{

  C_stack pop_()
  {
    return Sort_nothing;
  }

  C_item top_()
  {
    return Sort_nothing;
  }

};
C_empty-stack Sort_empty-stack = new C_empty-stack;


class C_natural : virtual public C_item
{

  C_natural sucessor_()
  {
    if ((this)->ISSuperSorte(Sort_natural))
```

```
    return new Co_sucessor_natural( (this)->The_natural() ) ;
  }

  C_natural sum__(C_natural p1)
  {
   {
    C_natural local0 = NULL;

    if ((p1)->Match_sucessor_(local0))
     return ((this)->sum__(local0))->sucessor_();
   }
   if ((p1==Sort_0))
    return this;
   if (((this)->ISSuperSorte(Sort_natural)&&(p1)->ISSuperSorte(Sort_natural)))
    return new Co_sum_natural_natural ( (this)->The_natural() , (p1)->The_natural() ) ;
  }

  C_natural fib_()
  {
   {
    C_natural local0 = NULL;

    if ((this)->Match_sucessor_(local0))
     {
      C_natural local1 = NULL;

      if ((local0)->Match_sucessor_(local1))
       return ((local0)->fib_())->sum__((local1)->fib_());
     }
   }
   if ((this)->ISSuperSorte(Sort_natural))
    return new Co_fib_natural( (this)->The_natural() ) ;
  }

  C_natural The_natural()
  {
   return this;
  }

};

class Ci_natural : virtual public C_natural
{

};
C_natural Sort_natural = new Ci_natural;


class C_0 : virtual public C_gen-s7
{
```

```
  bool Match_fib_(C_natural & param0)
  {
    param0 = Sort_0;
    return true;
  }

  bool Match_sucessor_()
  {
     return Sort_sucessor_0;
  }

};
C_0 Sort_0 = new C_0;


class C_nothing : C_stack , C_natural
{

};

C_nothing Sort_nothing = new C_nothing;

class Co_push_item_stack :  virtual public C_stack
{
  C_item v0;
  C_stack v1;

  Co_push_item_stack(C_item param0,C_stack param1)
  {
    v0 = param0;
    v1 = param1;
  }

  bool Match_push__(C_item & param0,C_stack & param1)
  {
    param0 = v0;
    param1 = v1;
    return true;
  }

};

class Co_pop_stack : virtual public C_stack
{
  C_stack v0 = NULL;

  Co_pop_stack(C_stack param0)
  {
    v0 = param0;
```

```
  }


};

class Co_top_stack : virtual public C_item
{
  C_stack v0 = NULL;

  Co_top_stack(C_stack param0)
  {
    v0 = param0;
  }


};

class Co_sucessor_natural : virtual public C_natural
{
  C_natural v0 = NULL;

  Co_sucessor_natural(C_natural param0)
  {
    v0 = param0;
  }
};

class Co_sucessor_0 : virtual public Co_sucessor_natural
{
  Co_sucessor_0(C_natural param0)
  {
    v0 = param0;
  }
  C_natural fib_()
  {
    return Sort_sucessor_0;
  }
};

C_natural Sort_sucessor_0 = new Co_sucessor_0(Sort_0);

class Co_sum_natural_natural : virtual public C_natural
{
  C_natural v0 = NULL;
  C_natural v1 = NULL;

  Co_sum_natural_natural(C_natural param0,C_natural param1)
  {
    v0 = param0;
    v1 = param1;
```

```
  }


};

class Co_fib_natural : virtual public C_natural
{
  C_natural v0 = NULL;

  Co_fib_natural(C_natural param0)
  {
    v0 = param0;
  }
};
```

Apêndice B

Especificação Algébrica da Notação de Ações

As próximas seções fornecerão a definição da notação de ações que é usada pelo sistema ABACO para produção de uma implementação da notação de ações. A especificação vista aqui é derivada da definição de semântica de ações mostrada em [Mos92]. Uma vez definida a especificação descreveremos algumas características do código fonte gerado pelo tradutor de especificações algébricas a partir da especificação de ações.

B.1 Notação de Ações

introduces: action, yielder,

includes: Notação de Dados,
Produtores, Faceta Basica,
Faceta Funcional,
Faceta Declarativa,
Faceta Reflexiva,
Faceta Imperativa.

- action $\geq \Box$.

B.1.1 Produtores

introduces: yielder.

needs: DataNotation.

(1) yielder $\geq$ data.

(2) *dataoperation* :: yielder, yielder,... $\rightarrow$ yielder.

O sorte yielder, que define os produtores na notação de ações, é um supersorte do sorte data.

O axioma 2, indica (informalmente) que todas as operações que manipulem indivíduos do sorte data, são expandidas para suportar produtores. Esse axioma por motivos óbvios não pode ser tratado pelo tradutor de especificações algébricas. Por isso o programador terá que incluir operadores como:

- sum _ _ :: yielder, yielder $\rightarrow$ yielder.

para que o modelo processe corretamente especificações em semântica de ações.

B.1.2 Faceta Básica

introduces: complete, fail, diverge, _ and _, _ andthen _, _ or _, unfolding _, unfold, escape, _ trap _, action.

- complete : action.
- fail : action.
- diverge : action.
- unfold : action.
- escape : action.
- _ and _ :: action, action $\rightarrow$ action.
- _ or _ :: action, action $\rightarrow$ action.

- _ andthen _ :: action, action → action.
- unfolding _ :: action → action.
- _ trap _ :: action, action → action.

B.1.3 Faceta Funcional

introduces: escape with _, _ then _, give _, action,yielder, given _, given _ # _.

- give _ :: yielder → action.
- _ then _ :: action, action → action.
- the given _ :: data → yielder.
- the given _ # _ :: datum, positive-integer → yielder.

B.1.4 Faceta Declarativa

introduces: bind _ to _, recursively bind _ to _, _ before _,
_ hence _, _ moreover _,
rebind, furthermore _, the _ bound to _,
bindable, token, bindings.

- bindable = □ .
- token = □ .
- bindings = map [token to bindable].
- bind _ to _ :: token, yielder → action.
- recursively bind _ to _ :: token, yielder → action.
- _ before _ :: action, action → action.
- _ hence _ :: action, action → action.
- _ moreover _ :: action, action → action.
- furthermore _ :: action → action.
- the _ bound to _ :: bindable, token → yielder.

B.1.5 Faceta Reflexiva

introduces: enact _, abstration of _, abstration, closure _.

- abstration $\leq$ datum.
- abstration of _ :: action $\rightarrow$ abstration.
- closure _ :: abstration $\rightarrow$ abstration.
- enact _ :: yielder $\rightarrow$ action.

B.1.6 Faceta Imperativa

introduces: allocate-a-cell, deallocate _, store _ in _, the _ stored in _, cell, storable, unitialized, storage.

- cell = $\square$.
- storable = $\square$.
- storage = map [cell to (storable **|** uninitialized)].
- allocate-a-cell : action.
- deallocate _ :: yielder $\rightarrow$ action.
- store _ in _ :: yielder, yielder $\rightarrow$ action.
- the _ stored in _ :: storable, yielder $\rightarrow$ yielder.

B.2 Implementação Gerada pelo Tradutor

O tradutor de especificações algébricas descrito no Capítulo 3 foi utilizado para produzir um código orientado a objetos que implementasse a especificação de ações. O código foi gerado com as seguintes características principais:

- Foram definidas uma classe de nome `C_action` cujas subclasses representarão as ações válidas. Foram gerados os seguintes tipos de subclasses.

1. Uma subclasse da classe `C_action` de nome `Ci_action` cujo único objeto, de nome `Sort_action`, é utilizado para representar o sorte action em expressões.

2. Cada subsorte do sorte action gerará uma classe e objeto para representá-lo. Como exemplo, tomaremos o sorte complete, para esse sorte o tradutor gerará o seguinte código.

```
class C_complete : C_action
{
  *******  Metodos da Classe *********
};
C_complete *Sort_complete = new C_complete;
```

3. Cada operador cujo resultado seja um subsorte de action, gerará uma sub-classe da classe `C_action` para representar seus elementos. Como exemplo seja o operador:

   - _ or _ :: action, action $\rightarrow$ action.

   gerará a seguinte classe:

```
class Co_(_or_)(action)(action) : C_action
{
   C_action v0;
   C_action v1;
   *******  Metodos da Classe *********
};
```

   O significado do nome classe `Co_(_or_)(action)(action)`, significa a classe que representa os resultados do operador _ or _ cujos parâmetros são subsortes do sorte action.

4. Cada operador que receba como primeiro parâmetro indivíduos do sorte action, produzirá métodos para a classe `C_action` e, se necessário, suas subclasses. Como exemplo, o operador _ or _ produz um método de nome `_or_` para a classe `C_action` que possuirá o seguinte esqueleto:

```
class C_action : Sort
```

```
{
  ......
  C_action _or_(C_action p1)
  {
    ***** Testa os operandos *******
    ***** Cria um objeto e retorna *****
  }
  ......
};
```

expressões de sorte como a_0 or a_1, são traduzidas para a seguinte expressão em C++ a_0`->_or_(`a_1`)`. O corpo do método `_or_` possui duas partes:

(a) Uma parte que testa os operandos para ver se eles correspondem a algum axioma. Essa parte é usada para implementar axiomas como:

fail or $x = x$.

(b) Uma parte responsável por criar um objeto da classe correspondente (no caso `Co_(_or_)(action)(action)`) e inicializá-lo com os parâmetros corretos.

- Os produtores (yielders) são implementados de forma análoga, sendo gerada uma classe de nome `C_yielder` cujas subclasses representarão todos os fornecedores existentes, as subclasses da classe `C_yielder` e as seguintes subclasses:

  1. Uma classe para representar o sorte yielder.
  2. Uma classe e objeto para cada elemento do sorte yielder.
  3. Uma classe para cada operador que gere um indivíduo subsorte de yielder e um método para gerar objetos dessa classe.

- O sorte bindings, definido como map [token to bindable], define uma subclasse da classe `C_map`, responsável pela definição de mapeamentos, para representar os mapeamentos que são subsortes de storage, o tradutor de especificações analisará a definição do operador map [_ to _] da notação de dados e definirá se um mapeamento deve ou não ser colocado como objeto subclasse da classe gerada por storage.

Apêndice C

Especificação Algébrica da Notação de Dados

Esse apêndice contém um subconjunto da notação de dados apresentada em [Mos92], que é utilizada pelo sistema ABACO.

C.1 Números

A especificação de números utilizada define duas entidades: números naturais e inteiros. As implementações desses sortes foram feitas utilizando C++, a especificação tem por utilidade permitir que os operadores possam ser usados por outras partes da especificação.

introduces: integer, negative-integer, predecessor _, negation _, component, natural, sucessor _, sum _ _, product _ _, difference _ _, division _ _, _ gt _, _ ge _, _ lt _, _ le _, _ is _.

- integer $\leq$ component.
- natural $\leq$ integer.
- predecessor _ :: integer $\rightarrow$ integer.
- sucessor _ :: integer $\rightarrow$ integer.
- negation _ :: integer $\rightarrow$ integer.

- sum _ _ :: integer, integer $\rightarrow$ integer.
- difference _ _ :: integer, integer $\rightarrow$ integer.
- product _ _ :: integer, integer $\rightarrow$ integer.
- division _ _ :: integer, integer $\rightarrow$ integer.
- _ gt _ :: integer, integer $\rightarrow$ truth-value.
- _ ge _ :: integer, integer $\rightarrow$ truth-value.
- _ lt _ :: integer, integer $\rightarrow$ truth-value.
- _ le _ :: integer, integer $\rightarrow$ truth-value.
- _ is _ :: integer, integer $\rightarrow$ truth-value.

C.2 Strings

De forma análoga aos números, a especificação define a classe string é feita diretamente em C++, ficando a especificação apenas para prover a interface com os operadores.

introduces: string, _ concat _.

needs: DataNotation.

- string $\leq$ component.
- _ concat _ :: string, string $\rightarrow$ string.

C.3 Valores Verdade

introduces: truth-value, true, false, component, both _ _, either _ _ , _ is _, if _ then _ else _.

- truth-value $\leq$ component.
- truth-value = true **|** false.
- _ is _ :: truth-value, truth-value $\rightarrow$ truth-value.

(1) true is true = true.

(2) false is false = true.

(3) true is false = false.

(4) false is true = false.

- both _ _ :: truth-value , truth-value $\rightarrow$ truth-value.

(5) both true x = x.

(6) both false x = false.

- either _ _ :: truth-value, truth-value $\rightarrow$ truth-value.

(7) either false x = x.

(8) either true x = true.

- not _ :: truth-value $\rightarrow$ truth-value.

(9) not true = false.

(10) not false = true.

- if _ then _ else _ :: truth-value, datum, datum $\rightarrow$ datum.

(11) if true then x else y = x.

(12) if false then x else y = y.

C.4 Tuplas

introduces: tuple, component, (), (_ _), $\_^*$, $\_^+$, $\_^?$, $\_^-$.

needs: DataNotation.

- component = $\Box$.
- component $\geq$ tuple.
- (_ _) :: tuple, tuple $\rightarrow$ tuple.
- ():tuple.

- $\_^*$ :: tuple → tuple.
- $\_^+$ :: tuple → tuple.
- $\_^?$ :: tuple → tuple.
- $\_^-$ :: tuple, natural → tuple.

C.5 Listas

C.6 Mapeamentos

introduces: map, map [_ to _], map _ to _, _ at _,
overlay _ _, disjoin-union _ _,
empty-map,range, element.

needs: DataNotation.

- map [_ to _] :: range, element → map.
- map _ to _ :: range, element → map.
- empty-map : map.
- _ at _ :: map, range → element.
- overlay _ _ :: map, map → map.
- disjoin-union _ _ :: map, map → map.
- range = □ .
- element = □ .

C.7 Data

introduces: data, datum.

needs: DataNotation.

- datum $\leq$ component.
- datum $\geq$ integer **|** truth-value **|** string.
- data = datum *.
- bindings = map [token to bindable].
- storage = (integer, map [cell to storable]).

C.8 Sintaxe Abstrata

introduces: abstract-syntax, $\langle\ \_\ \rangle^?$, $\langle\ \_\ \rangle^*$, $[\![]\!]$, $[\![\ \_\ ]\!]$, ...

needs: DataNotation/String.

- string $\leq$ abstract-syntax.
- $[\![]\!]$: abstract-syntax.
- $[\![\ \_\ ]\!]$ :: abstract-syntax $\rightarrow$ abstract-syntax .
- $[\![\ \_ \ldots \_\ ]\!]$ :: abstract-syntax, ..., abstract-syntax $\rightarrow$ abstract-syntax.
- $\langle\ \_\ \rangle^?$ :: abstract-syntax $\rightarrow$ abstract-syntax.
- $\langle\ \_\ \rangle^*$ :: abstract-syntax $\rightarrow$ abstract-syntax.

(1) $\langle\ x\ \rangle^? = [\![]\!]$ **|** x.

(2) $\langle\ x\ \rangle^* = [\![]\!]$ **|** $[\![\ x\ \langle\ x\ \rangle^*\ ]\!]$.

Apêndice D

Especificação Dinâmica da Notação de Ações

Esse apêndice fornecerá uma descrição formal do processo de execução de uma ação, segundo o especificado em [Mos92]. Nessa especificação definiremos a semântica do operador perform _, que simulará o processo de execução de uma ação.

D.1 Ambiente de Execução

introduces: info, resul, state.

needs: DataNotation.

- info = (data, bindings, storage, truth-value, action).
- resul = (state, info).
- state = completing | failing | escaping.

O sorte info, responsável por modelar o estado das informações correntes é formado por uma tupla com os seguintes elementos:

- data: responsável por modelar as informações transitórias.

- bindings: responsável por modelar as associações existentes em um programa.
- storage: responsável por modelar a memória.
- truth-value: responsável por registrar o comprometimento de storage pela ação.
- action: armazena a ação **unfolding** que está sendo executada. Caso nenhuma ação unfolding esteja sendo executada o valor desse campo é nothing.

O resultado da execução de uma ação é dado pelo sorte resul que é composto por uma tupla contendo um estado de término produzido pela ação e as informações produzidas pela execução da ação.

Os estados de término (outcome), definidos pela execução são: completing, failing e escaping.

D.2 Operador perform _ _

O operador perform _ _ é utilizado para a execução da ação. A expressão perform a i irá executar a ação a passando as informações contidas em i. O resultado da execução da ação é devolvido pelo operador na forma de um indivíduo do sorte resul.

introduces:

- perform _ _ :: action, info → resul.

(1) perform complete (d,b,s,u) = (completing, ((),empty-map,s,false,u)).

(2) perform fail (d:data,b:bindings,s,u) = (failing, ((),empty-map,s, false, u)).

(3) perform escape (d,b,s,u) = (escaping, (d,empty-map,s, false, u)).

(4) d = evaluate y i $\Rightarrow$
perform (escape with y) i:(d,b,s,u) = (escaping, (d,empty-map,s, false, u))

(5) t = evaluate y i $\Rightarrow$
perform (give y) i:(t,b,s,c,u) =
if (t=nothing) then (failing,((),empty-map,s,false, u))
else (completing,(t,empty-map,s,false, u)).

(6) v = evaluate y i $\Rightarrow$
perform (bind t to y) i:(t,b,s,c, u) =
if (v=nothing) then (failing,((),empty-map,s,false, u))
else (completing,((),map t to v,s,false, u)).

(7) c = evaluate y_2 i $\Rightarrow$
d = evaluate y_1 i $\Rightarrow$
f = either (c=nothing) (d=nothing) $\Rightarrow$
perform (store y_1 in y_2) i:(t,b,s,c, u) =
if (f) then (failing,((),empty-map,s,false, u))
else (completing,((),empty-map,overvaly (map c to b, s),true, u)).

(8) v = evaluate y i $\Rightarrow$
perform (escape with y) i:(t,b,s,c,u) =
if (v=nothing) then
(failing,((),empty-map,s,false, u))
else (escaping,(v,empty-map,s,false, u)).

(9) s at c = nothing $\Rightarrow$
perform allocate-a-cell i:(t,b,s,c,u) =
(completing,((c),empty-map, overlay(map c to uninitialized, s),true, u)).

(10) c = evaluate y i $\Rightarrow$
perform deallocate y i:(t,b,s,c,u) =
(completing,((),empty-map,overlay (map c to nothing, s),true, u)).

(11) perform commit i:(t,b,s,u) = (completing,((),empty-map,s,true, u));

(12) perform (a and b) i = step i (perform a i) (and) b.

(13) perform (a andthen b) i = step i (perform a i) (andthen) b.

(14) perform (a or b) i = step i (perform a i) (or) b.

(15) perform (a trap b) i = step$i$ (perform a i) (trap) b.

(16) perform (a then b) i = step i (perform a i) (then) b.

(17) perform (a moreover b) i = step i (perform a i) (moreover) b.

(18) perform (a hence b) i = step i (perform a i) (hence) b.

(19) perform (a before b) i = step i (perform a i) (before) b.

(20) perform (futhermore a) i = join i (futhermore) (perform a i).

(21) perform (unfolding a) i:(t,b,s,c,u) = perform a (t,b,s,c,a).

(22) perform (unfold) i:(t,b,s,c,u) = perform u i.

(23) abstration of a = evaluate y i $\Rightarrow$
perform (enact y) i = perform a i.

(24) closured b_2 abstration of a = evaluate y i $\Rightarrow$
perform (enact y) i:(t,b_1,s,c,u) = perform a (t,b_2,s,c,u).

A execução de uma ação se dá da seguinte forma:

- Ações primitivas: são executadas de acordo com o seu significado.
- Ações combinadas:
  - Combinadores and, andthen, or, trap, then, moreover, hence, e before: A primeira subação é executada e o resultado dessa execução, juntamente com a segunda subação e o combinador são passados para o operador step que decidirá qual o próximo passo para execução da ação.
  - Combinador unfolding: é executado processando-se a sua subação com as mesmas informações recebidas pelo combinador com a exceção que o campo indicador da ação unfolding corrente é alterado.
  - Ação unfolding: ao ser executada, essa ação executa a ação unfolding corrente.

D.3 Operadores Auxiliares

D.3.1 Operador step _ _ _ _

- op-infix = op-seq | op-or | trap | futhermore.
- op-seq = andthen | then | and | moreover | hence | before.
- step _ _ _ _ :: info, resul, op-infix, action → resul.

(1) o : (failing | escaping) $\Rightarrow$
step i_1 r:(o,i_2) s:op-seq a = r.

(2) step (t_1,b_1,s_1,c_1,u_1) (completing, (t_2,b_2,s_2,c_2,u_2)) then a = join then r (perform a (t_2, b_1, s_1,c_1,u_1)).

(3) o : (and | andthen) $\Rightarrow$
step (t_1,b_1,s_1,c_1,u_1) (completing, (t_2,b_2,s_2,c_2,u_2)) o a =
join o r (perform a (t_1, b_1, s_2,c_2,u_1)).

(4) step (t_1,b_1,s_1,c_1,u_1) (completing, (t_2,b_2,s_2,c_2,u_2)) moreover a =
join moreover r (perform a (t_2, b_1, s_2,c_2,u_1)).

(5) step (t_1,b_1,s_1,c_1,u_1) (completing, (t_2,b_2,s_2,c_2,u_2)) hence a =
join hence r (perform a (t_2, b_2, s_2,c_2,u_1)).

(6) step (t_1,b_1,s_1,c_1,u_1) (completing, (t_2,b_2,s_2,c_2,u_2)) before a =
join before r (perform a (t_2, overlay b_2 b_1, s_2,c_2,u_1)).

(7) o : (completing | escaping) $\Rightarrow$
step (t_1,b_1,s_1,c_1,u_1) r:(o, (t_2,b_2,s_2,c_2,u_2)) or a = r.

(8) $s$tep i_1:(t_1,b_1,s_1,c_1,u_1) i_2:(failing, (t_2,b_2,s_2,c_2,u_1)) or a =
if (c_1) then i_2 else (perform a i_1).

(9) o : (completing | failing) $\Rightarrow$
step (t_1,b_1,s_1,c_1,u_2) r:(o, (t_2,b_2,s_2,c_2,u_2)) trap a = r.

(10) step (t_1,b_1,s_1,c_1,u_1) r:(escaping, (t_2,b_2,s_2,c_2,u_2)) trap a =
perform a (t_2, b_1,s_2,c_2,u_1).

O operador step recebe como parâmetros:

- As informações recebidas pela ação combinada.
- As informações produzidas por uma das subações.
- O combinador de ações
- A segunda subação que deve ser executada.

Durante a chamada ao operador step, este verificará o resultado da primeira ação executada e tomará uma das seguintes providências, dependendo do combinador e do resultado da execução da primeira ação:

- Retorna o resultado produzido pela primeira ação, se o resultado da execução desta indicar que a segunda ação não deve ser executada.
- Executa a segunda ação, com as informações produzidas pela primeira ação e recebidas pelo combinador, combinadas de acordo com a semântica do combinador e calcula as informações produzidas pela ação combinada baseando-se nas informações recebidas pelo combinador e as informações produzidas pela execução das subações. Em alguns casos em que esse cálculo é mais complexo é chamado o operador join _ _ _, responsável pelo cálculo das informações produzidas pelo combinador.

D.3.2 Operador join _ _ _

- join _ _ _ :: info, op-infix, resul → resul.

(1) o : (failing | escaping) ⇒
join i_1 op r_2:(o,i_2) = r_2.

(2) join (t_1,b_1,s_1,c_1,u_1) then (completing,(t_2,b_2,s_2,c_2,u_2)) = (t_2, overlay b_1 b_2, s_2, either c_1 c_2, u_1).

(3) join (t_1,b_1,s_1,c_1,u_1) andthen (completing,(t_2,b_2,s_2,c_2,u_2)) = ((t_1,t_2), overlay b_1 b_2, s_2, either c_1 c_2, u_1).

(4) join (t_1,b_1,s_1,c_1,u_1) and (completing,(t_2,b_2,s_2,c_2,u_2)) = ((t_1,t_2), overlay b_1 b_2, s_2, either c_1 c_2, u_1).

(5) join (t_1,b_1,s_1,c_1,u_1) moreover (completing,(t_2,b_2,s_2,c_2,u_2)) = ((t_1,t_2), overlay b_2 b_1, s_2, either c_1 c_2,u_1).

(6) join (t_1,b_1,s_1,c_1,u_1) hence (completing,(t_2,b_2,s_2,c_2,u_2)) = (($t_1 t_2$), b_2, s_2, either c_1 c_2, u_1).

(7) join (t_1,b_1,s_1,c_1,u_1) before (completing,(t_2,b_2,s_2,c_2,u_2)) = ((t_1,t_2), overlay b_2 b_1, s_2, either c_1 c_2, u_1).

(8) join (t_1,b_1,s_1,c_1,u_1) (futhermore) (completing,(t_2,b_2,s_2,c_2,u_2)) = (t_2, overlay b_2 b_1,s_2, c_2, u_1).

A função do operador join, consiste em receber as informações produzidas pelas subações de uma ação combinada e gerar as informações produzidas pelo combinador de acordo com a semântica deste.

D.3.3 Operador evaluate _ _

introduces: evaluate _, closured _ _.

- closured _ _ :: bindings, abstration → abstration.
- evaluate _ _ :: yielder, info → data.

(1) x : data $\Rightarrow$ evaluate x i = x.

(2) evaluate (sum (x y)) i = sum (evaluate x i) (evaluate y i).

(3) .....

(4) evaluate (given x) i:(t,b,s) = if ($t \leq x$) then t else nothing.

(5) v = component n t $\Rightarrow$
evaluate (given x # n) i:(t,b,s,c,u) = if ($v \leq x$) then v else nothing.

(6) v = b at t $\Rightarrow$
evaluate (the x bound t) i:(t,b,s,c,u) = if ($v \leq x$) then v else nothing.

(7) c = evaluate y i $\Rightarrow$
v = s at c $\Rightarrow$
evaluate (the x stored in y) i:(t,b,s,c,u) = if ($v \leq x$) then v else nothing.

(8) evaluate (closure x) i:(t,b,s,c,u) = closured b (evaluate x i).

A função do operador evaluate consiste em calcular o valor dos produtores que dependem das informações correntes.

O operador closured, definido nessa especificação, é usado parta representar expressões que usem o operador closure depois de avaliadas.

D.4 Implementação

A implementação de todo o sistema interpretador de ações, como está mostrado nesse apêndice, não é viável pelo método de tradução mostrada nessa tese. O principal motivo para isso é o axioma que indica a forma de execução da ação allocate-a-cell:

> s at c = nothing $\Rightarrow$
> perform allocate-a-cell i:(t,b,s,c,u) = (completing,((c),empty-map, overlay(map c to uninitialized, s),true, u)).

esse axioma indica que deve-se escolher, de forma não determinista, uma célula da memória que não esteja alocada e aloca-la. O tradutor não tem condições para decidir o resultado de operações desse tipo. Outro fator que impediria a especificação é o fato de que células de memória não são definidas precisamente pela notação de ações.

Para suplantar essas dificuldades é necessário definirmos qual o significado do sorte cell uma alternativa obvia seria a sua definição como subsorte dos números naturais. Outra alteração nesse axioma seria fornecer uma especificação precisa sobre o processo de alocação de memória como por exemplo a célula alocada seria o primeiro número natural não alocado.

Um outro fator que impediu a implementação literal das execução da notação de ações foi a impossibilidade do cálculo de expressões como:

> if ($v \leq x$) then v else nothing

devido ao fato da implementação não prever o uso dos operadores de sorte dentro de expressões. Entretanto nada no método impede que isso seja implementado, apenas não esta sendo suportada na versão corrente do tradutor.

Como exemplo de código gerado, o operador perform _ gera um método na classe `C_action` e em suas subclasses. Essas classes foram definidas pela especificação de ações vista no Apêndice B. O corpo desse método é definido pelos axiomas. Como exemplo, o axioma:

(1) perform fail (d:data,b:bindings,s,u) = (failing, ((),empty-map,s, false, u)).

gera o seguinte trecho de código no método `perform_` da classe `Ci_fail`, responsável por implementar o sorte fail:

```
C_resul perform_(C_info p1)
{
  .....
  {
    C_data local0;
    C_bindings local1;
    C_storage local2;
    C_truth-value local3;
    if (p1->Match_(____)(local0,local1,local2,local3))
    {
      return new C_(__)(Sort_failing,
                        new C_(____)(Sort_(),
                                     Sort_empty-map,
                                     local2,local3
                                    )
                       );
    }
  }
  ......
}
```

onde:

- `C_(____)`: é a classe responsável por definir tuplas.
- `Match_(____)`: é uma função que retorna `true` se o objeto é um formado por uma tupla.