Lógica de Reescrita e Maude

Especificações em Semântica Operacional Executáveis

Christiano Braga
cbraga@ic.uff.br

http://www.ic.uff.br/~cbraga

Universidade Federal Fluminense

Agradecimentos

- A semântica de reescrita de bc, parte desta palestra, é fruto de trabalho conjunto com o Prof. José Meseguer, da Universidade de Illinois em Urbana-Champaign nos EUA.
- Gostaria de agradecer a organização do SBLP e da Escola de Linguagens de Programação pela oportunidade de apresentar este mini-curso.
- Ao CNPq e a FAPERJ pelo auxílio parcial para o desenvolvimento deste trabalho.

PG no DCC da UFF

- Mestrado e Doutorado
- 22 docentes nas áreas:
  - Métodos Formais

    `http://gmf.ic.uff.br`
  - Otimização Combinatória e I.A.
  - Processamento Distribuído e Paralelo
  - Computação Visual e Interfaces
  - Modelagem Computacional
  - Computação em Potência

PG no DCC da UFF

- IDH de Niterói é 0,886 sendo o melhor do estado do Rio e o 3º do país

  `http://www.niteroi.rj.gov.br/`

- `http://www.ic.uff.br/PosGrad`

Objetivo

O objetivo deste mini-curso é mostrar como especificações em semântica operacional podem ser executadas na linguagem Maude.

Agenda

1. Semântica formal
2. Abordagens
3. Semântica operacional
4. Modularidade em semântica operacional
5. Estudo de caso: bc
6. Semântica operacional de bc

Agenda

7. Lógica de reescrita
8. Maude
9. Semântica de reescrita
10. Semântica de reescrita modular para bc

Semântica formal

- Semântica formal de uma linguagem de programação é uma especificação matemática do *significado* da sua sintaxe.
- O significado depende na realidade da abordagem utilizada. Podemos pensar por exemplo numa função que “implementa” a semântica da linguagem.

Semântica formal

Exemplo: o significado da atribuição `x = 10` pode ser vista como a aplicação da função $eval$ cuja assinatura é:

$$eval : SintaxeMemoria \rightarrow Memoria$$

a atribuição `x = 10` e a uma memória (conjunto de pares variável-valor)

$$eval(\texttt{x} = \texttt{10}, \{(a, 1), (s, \text{“}ola'')\})$$

produzindo então $\{(a, 1), (s, \text{“}ola''), (x, 10)\}$.

Semântica formal

- A função $eval$ faz parte de uma especificação para uma determinada linguagem de programação, digamos, L.
- A função $eval$ é uma *função semântica*, que especifica o significado de todas os possíveis programas em L.

Semântica formal

Ao invés de especificar a sintaxe de cada possível programa um-a-um, uma especificação semântica descreve de maneira genérica o significado de todos os (sub)programas que possuem uma determinada sintaxe.

Semântica formal

Uma *gramática* especifica todos os programas possíveis numa dada linguagem L. Uma gramática para uma linguagem simples com as operações aritméticas e mais a atribuição poderia ser escrita da seguinte maneira:

PROG	:=	EXP | CMD | PROG ; PROG
EXP	:=	ID | EXP OP EXP
OP	:=	+ | - | * | /
CMD	:=	ID = EXP | noop
ID	:=	[0..9]+

Semântica formal

- Uma especificação da semântica formal de uma linguagem de programação L relaciona a gramática de L com elementos matemáticos como por exemplo conjuntos. Neste caso utilizaríamos a *teoria de conjuntos* para dar a semântica de uma linguagem de programação.
- Existem então diferentes *abordagens* para a especificação formal da semântica de linguagens de programação. Vejamos então algumas delas.

Abordagens

Faremos uma rápida passagem sobre as seguintes abordagens para a especificação formal de linguagens de programação:

1. Semântica denotacional
2. Semântica axiomática
3. Semântica algébrica
4. Semântica operacional

Abordagens

Semântica denotacional

- Criada por Christopher Strachey em meados de 1960. A fundamentação matemática foi dada por Dana Scott em 1969.
- O significado da linguagem é dado em termos de um mapeamento para objetos matemáticos como funções e números. Por isso chamava-se inicialmente semântica matemática.

Abordagens

Semântica denotacional

- O exemplo da atribuição utiliza esta abordagem. Diz-se então que a (aplicação da) função $eval$ é a denotação da atribuição.
- Um aspecto muito importante desta abordagem é o fato dela ser composicional, ou seja, a denotação (significado) de uma construção sintática é dada em termos da denotação de seus componentes.

Abordagens

Semântica denotacional

- No caso da atribuição é necessário saber qual a denotação da expressão (do lado direito da atribuição) a ser associada a variável (do lado esquerdo da atribuição).
- $eval(\mathtt{V} = \mathtt{E}, M) = M \cup \{(V, eval(E, M)\}$

Abordagens

Semântica axiomática

- Criada (paralelamente) por Robert Floyd e C.A.R. Hoare.
- Baseada em assertivas lógicas que devem ser verdadeiras a cada execução de um programa.

Abordagens

Semântica axiomática

- Estas assertivas são:
  - pré-condição: assertiva que deve ser válida antes da execução de um programa.
  - pós-condição: assertiva que deve ser válida após a execução de um programa.
  - invariante: assertiva que deve ser válida a qualquer momento da execução de um programa.

Abordagens

Semântica axiomática

- A especificação para a atribuição é dada pelo seguinte axioma:

$$\{P(E)\}\ V = E\ \{P(V)\}$$

onde P é um predicado (expressão booleana).

Abordagens

Semântica axiomática

- O axioma significa que se é possível provar uma propriedade P a respeito da expressão E antes da atribuição, então esta mesma propriedade é verdade para V após a atribuição.

Abordagens

Semântica algébrica

- Desenvolvida por diversos pesquisadores como Joseph Goguen e José Meseguer.
- A semântica da linguagem de programação é dada em termos de tipos abstratos de dados (sorts) através de equações.
- O significado de um (sub)programa é dado pela aplicação das equações substituindo “iguais por iguais”.

Abordagens

Semântica algébrica

- No exemplo da atribuição a memória é representada pelo sort Mem que possui operações incluindo:

$$sorts\ Cel\ Mem\ .$$
$$cel : Var\ Valor\ \rightarrow Cel\ .$$
$$memVazia :\ \rightarrow Mem\ .$$
$$mem : Cel\ Mem\ \rightarrow Mem\ .$$
$$atualizaMem : Mem\ Cel \rightarrow Mem\ .$$

Abordagens

Semântica algébrica

- As operações cel, $memVazia$ e mem são chamadas de *construtores*.
- A expressão $cel(x, 1)$ é dita um *termo* na *álgebra* definida pelos operadores e pelas declarações de sort do slide anterior.

Abordagens

Semântica algébrica

- O conjunto de operações mais o conjunto de declarações de sorts define a assinatura de uma especificação. A álgebra definida por uma assinatura Σ é chamada de Σ-álgebra.
- Em especial, a álgebra dos termos, ou álgebra inicial de uma assinatura Σ é representada por T_Σ.

Abordagens

$$
\begin{aligned}
&atualizaMem(\\
&\quad mem(cel(V : Var, L : Valor), M : Mem), \\
&\quad cel(V' : Var, L' : Valor)) = \\
&if\ V == V'\ then \\
&\quad mem(cel(V : Var, L' : Valor), M : Mem) \\
&else \\
&\quad mem(cel(V : Var, L : Valor), \\
&\qquad atualizaMem(M : Mem, \\
&\qquad\quad cel(V' : Var, L' : Valor))) .
\end{aligned}
$$

Abordagens

Semântica algébrica

$$atualizaMem(memVazia, C : Cel) = mem(C : Cel, memVazia) \ .$$

$$eval(V : Var = E : Expr, M : Mem) = atualizaMem(M : Mem, cel(V : Var, eval(E : Expr, M : Mem))) \ .$$

Abordagens

Semântica operacional

- Desenvolvida simultaneamente por Gordon Plotkin e Gilles Khan, porém cada um com uma técnica um pouco diferente.
- Construções sintáticas são mapeadas para objetos matemáticos utilizando regras de inferência, como em lógica.
- Vamos estudar mais detalhadamente este formalismo.

Semântica operacional

- Em semântica operacional são utilizadas *regras de inferência* para especificar que computações são realizadas para cada construção sintática da linguagem.
- A avaliação da soma poderia ser especificada pela seguinte regra de inferência:

(1)
$$\frac{E_1 \rightarrow V_1 \wedge E_2 \rightarrow V_2 \wedge V = V_1 +_{\mathbb{Z}} V_2}{E_1 + E_2 \rightarrow V}$$

Semântica operacional

- A regra 1 significa que a soma das expressões E_1 e E_2 se dá avaliando-se (simultaneamente!) E_1 e E_2 sendo que o resultado final é dado pela soma dos inteiros produzidos pela avaliação de E_1 e E_2.
- Pode-se também fazer uma leitura desta regra sob a ótica da lógica: diz-se então que é possível *deduzir* V a partir de E_1+E_2 se for possível deduzir V_1 de E_1 e V_2 de E_2.

Semântica operacional

O *modelo* (ou significado) de uma especificação S em semântica operacional é um sistema de transição $(E, F, \rightarrow)$ onde E é o conjunto de estados do sistema de transição (como por exemplo `x = 10`), $F \subset E$ é o conjunto de estados finais do sistema de transição, e $\rightarrow \subseteq E/F \times E$ é a relação de transição, onde E/F significa o conjunto de estados *modulo* os estados finais, ou seja, "a menos" dos estados finais.

Semântica operacional

- Uma computação é precisamente uma sequência de transições *adjacentes*, ou seja, o estado destino da primeira transição é o estado de origem da segunda.
- Os estados também são chamados de configurações.

Semântica operacional

Classicamente existem duas técnicas para a especificação das regras de inferência:

1. a semântica operacional estrutural (SOS–structural operational semantics), ou “small-step”, ou ainda semântica de computação, proposta por Gordon Plotkin, onde as regras de inferência especificam “passo-a-passo” como uma computação evolui.

Semântica operacional

2. a semântica natural (NS–natural semantics), ou “big-step”, ou ainda semântica de avaliação, proposta por Gilles Khan, onde as regras de inferência especificam *em um passo* como a computação se dá.

Semântica operacional

- A regra 1 é um exemplo de regra em NS.
- Em SOS:

(2)
$$\frac{E_1 \rightarrow E_1'}{E_1 + E_2 \rightarrow E_1' + E_2}$$

(3)
$$\frac{E_2 \rightarrow E_2'}{v + E_2 \rightarrow v + E_2'}$$

(4)
$$\frac{V = V_1 +_{\mathbb{Z}} V_2}{V_1 + V_2 \rightarrow V}$$

Semântica operacional

- Como já vimos anteriormente a semântica do comando de atribuição é dada pela inserção de um novo par variável-valor numa representação da memória.
- “Classicamente” a memória é representada como parte da configuração.

(5)
$$\frac{M \vdash E \rightarrow L}{M \vdash V = E \rightarrow M \cup (V, L) \vdash noop}$$

Semântica operacional

- Essa representação, no entanto, não é *modular*. A estrutura da configuração foi modificada pelo uso da sintaxe $\vdash$.
- A regra para a soma de expressões por exemplo precisaria ser *redefinida* de forma a considerar a memória apesar não fazer uso dela.
- Vejamos como especificações em semântica operacional podem ser escritas de maneira modular.

Modularidade em SO

- Tradicionalmente as transições são elementos ternários e não binários como vimos até agora. O terceiro elemento é chamado de rótulo (label) e classicamente é utilizado para representar sinais de sincronização em sistemas concorrentes.
- A relação de transição é então definida como $\rightarrow \subseteq E \times L \times E$ onde L é o conjunto dos labels.

Modularidade em SO

- Recentemente Peter Mosses propôs uma abordagem para a definição de especificações modulares em semântica operacional fazendo uso de transições rotuladas.
- Mosses propõe um uso mais genérico dos rótulos em relação a abordagem clássica. Nos rótulos ficam armazenadas *quaisquer* estruturas semânticas, como por exemplo a memória.

Modularidade em SO

- O rótulo é então estruturado na forma de um registro (record) onde cada índice do record representa uma estrutura semântica diferente.
- A regra 5 fica então da seguinte forma:

$$(6)\quad \frac{\begin{array}{c} E \xrightarrow{\alpha} L \wedge \\ \alpha = set\_post(\alpha, mem, \\ atualiza(\alpha, get\_pre(\alpha, mem), (V, L))) \end{array}}{V = E \xrightarrow{\alpha} noop}$$

Modularidade em SO

- O índice mem possui um *par* de memórias representando a memória antes de uma transição e a memória depois da transição.
- O que a regra 6 especifica é que ocorre uma transição da configuração $V = E$ para $noop$ com o rótulo α, onde a memória após a atribuição é dada pela memória antes da atribuição porém atualizada com o par (V, L), onde L é o resultado da avaliação da expressão E.

Modularidade em SO

- O operador $noop$ é um comando que ”não faz nada”. Matematicamente é a identidade do operador ; para composição de comandos.

Estudo de caso: bc

- Para ilustrar o uso de semântica operacional modular vejamos uma especificação para a semântica da linguagem bc do projeto GNU.
- GNU bc (basic calculator) é definida como uma calculadora de precisão arbitrária e é usualmente distribuída junto com o sistema GNU/Linux. Possui uma sintaxe similar à linguagem C, porém, não é tipada. Possui um interpretador e um compilador para uma representação similar aos byte codes de Java.

Estudo de caso: bc

- O código a seguir é uma implementação para a função fatorial em bc.

```
define fat(x) {
 if (x == 0) {
   return(1) ;
 } else {
   return(x * fat(x - 1)) ;
 }
}
fat(20)
```

- Produz o resultado `2432902008176640000`.

Semântica operacional de bc

- Expressões aritméticas em bc são especificadas pelas seguintes regras em NS.

(7)
$$\frac{E_1 \xrightarrow{\alpha'} V_1 \ \wedge \ E_2 \xrightarrow{\alpha''} V_2 \ \wedge \ \alpha = \alpha';\alpha'' \ \wedge \ V = V_1 +_{\mathbb{Z}} V_2}{E_1 + E_2 \xrightarrow{\alpha} V}$$

(8)
$$\frac{E_1 \xrightarrow{\alpha'} V_1 \ \wedge \ E_2 \xrightarrow{\alpha''} V_2 \ \wedge \ \alpha = \alpha';\alpha'' \ \wedge \ V = V_1 -_{\mathbb{Z}} V_2}{E_1 - E_2 \xrightarrow{\alpha} V}$$

Semântica operacional de bc

(9)
$$\frac{E_1 \xrightarrow{\alpha'} V_1 \ \wedge \ E_2 \xrightarrow{\alpha''} V_2 \ \wedge \ \alpha = \alpha'; \alpha'' \ \wedge \ V = V_1 *_{\mathbb{Z}} V_2}{E_1 * E_2 \xrightarrow{\alpha} V}$$

(10)
$$\frac{E_1 \xrightarrow{\alpha'} V_1 \ \wedge \ E_2 \xrightarrow{\alpha''} V_2 \ \wedge \ \alpha = \alpha'; \alpha'' \ \wedge \ V_2 > 0 \ \wedge \ V = V_1 /_{\mathbb{Q}} V_2}{E_1 / E_2 \xrightarrow{\alpha} V}$$

Semântica operacional de bc

- A sintaxe de comandos em bc tem a seguinte gramática:

```
   CMD   :=   noop | ATRIB | COND | LOOP
 ATRIB   :=   ID = EXP
 COND    :=   if ( EXP ) BLOCO |
              if ( EXP ) BLOCO else BLOCO
 LOOP    :=   FOR | WHILE
  FOR    :=   for ( CMD ; EXP ; CMD ) BLOCO
WHILE    :=   while ( EXP ) BLOCO
```

Semântica operacional de bc

- A sintaxe de blocos é dada pelas seguintes regras:

BLOCO := CMD | { CMD-LIST }
CMD-LIST := CMD | CMD ; CMD-LIST

Semântica operacional de bc

- Já vimos como fica a regra para uma atribuição:

$$\text{(11)}\quad \frac{\begin{array}{c} E \xrightarrow{\alpha} L \wedge \\ \alpha = set\_post(\alpha, mem, \\ atualiza(\alpha, get\_pre(\alpha, mem), (V, L))) \end{array}}{V = E \xrightarrow{\alpha} noop}$$

Semântica operacional de bc

- Condicional:

(12)
$$\frac{E \xrightarrow{\alpha} true}{if\ (E)\ B_1\ else\ B_2 \xrightarrow{\alpha} B_1}$$

(13)
$$\frac{E \xrightarrow{\alpha} false}{if\ (E)\ B_1\ else\ B_2 \xrightarrow{\alpha} B_2}$$

(14)
$$if\ (E)\ B \xrightarrow{\alpha} if\ (E)\ B\ else\ noop$$

Semântica operacional de bc

- Loops:

$$\text{(15)}\quad \frac{E \xrightarrow{\alpha} V}{while\ (E)\ B \xrightarrow{\alpha} \begin{array}{l} if\ V == true\ then \\ \quad (B\ ;\ while\ (E)\ B)\ else\ noop\ fi \end{array}}$$

$$\text{(16)}\quad \frac{}{for\ (C_1\ ;\ E\ ;\ C_2)\ B \xrightarrow{\alpha} C_1\ ;\ while\ (E)\ \{B\ ;\ C_2\}}$$

$$\text{(17)}\quad \frac{C_1 \xrightarrow{\alpha} noop}{\{C_1; CL_2\} \xrightarrow{\alpha} CL_2}$$

Semântica operacional de bc

- Para a semântica de comandos só precisamos da memória.
- A semântica de declaração de funções precisa no entanto de um outro componente semântico: o ambiente de declarações.
- Para a linguagem bc o ambiente de declarações possui somente associações entre nome de função e o corpo da função.

Semântica operacional de bc

- A gramática para a declaração de uma função é a seguinte:

FUN	:=	define ID (ID-LIST) CORPO
ID-LIST	:=	nil | ID , ID-LIST
CORPO	:=	{ } | { auto ID-LIST ; } | BLOCO |
		{ auto ID-LIST ; CMD-LIST }

- A sintaxe `auto` declara uma lista de variáveis locais a função.

Semântica operacional de bc

(18) $$\frac{\alpha = set(\iota, env, (F, (PL, AL, CL))}{define\ F(PL)\ \{auto\ AL\ ;\ CL\} \xrightarrow{\alpha} noop}$$

- A regra 18 especifica que é criada no ambiente, referenciado pelo índice env, uma associação entre F e a tripla (PL, AL, CL). Esta tripla representa uma *abstração* da função, contendo sua lista de parâmetros formais, lista de variáveis locais e corpo da função.

Semântica operacional de bc

- Finalmente para especificar a semântica de uma chamada de função precisamos de uma outra estrutura semântica: uma pilha para armazenar os registros de ativação de cada chamada de função.
- Um registro de ativação precisa armazenar as associações entre os parâmetros formais (definidos na declaração) da função e os parâmetros atuais (definidos na chamada da função) assim como as associações das variáveis locais.

Semântica operacional de bc

- O primeiro passo é avaliar os parâmetros da chamada.

(19)
$$\frac{PL \xrightarrow{\alpha} VL}{F(PL) \xrightarrow{\alpha} apl(F, VL)}$$

Semântica operacional de bc

- Em seguinda precisamos empilhar o novo registro de ativação.

$$\frac{\begin{array}{c}(PL, AL, B) = \mathit{find}(\mathit{get}(\alpha, env), F) \ \wedge \\ K = \mathit{getStack}(\mathit{get\_pre}(\alpha, mem)) \ \wedge \\ AR = \{\mathit{initValList}(PL, VL), \mathit{initVal}(AL, 0)\} \ \wedge \\ \alpha = \mathit{set\_post}(\alpha, mem, \\ \mathit{setStack}(\mathit{get\_pre}(\alpha, mem), \mathit{push}(K, AR)))\end{array}}{\mathit{apl}(F, VL) \xrightarrow{\alpha} \mathit{call}(F, B)}$$

(20)

Semântica operacional de bc

- Ao final da chamada simplesmente desempilhamos o topo da pilha dos registros de ativação.

$$\text{(21)}\quad \frac{\begin{array}{c} B \xrightarrow{\alpha} V \ \wedge \\ K = getStack(get\_pre(\alpha, mem)) \ \wedge \\ \alpha = set\_post(\alpha, mem, \\ setStack(get\_pre(\alpha, mem), pop(K))) \end{array}}{call(F, B) \xrightarrow{\alpha} V}$$

Lógica de reescrita

- Desenvolvida por José Meseguer e seu grupo no SRI International em 1992.
- Proposta como um formalismo lógico-semântico para a qual é possível mapear lógicas e linguagens de especificação.
- Alguns exemplos são:
  - Lógica equacional, lógica de Horn, lógica linear.
  - Programação OO concorrente, CCS, SOS.

Lógica de reescrita

- Uma especificação ou teoria em lógica de reescrita possui um componente equacional, ou seja, especificado utilizando equações como as que vimos em especificações algébricas, e um componente especificado em termos de regras.
- Não seria possível cobrir todos os aspectos de lógica de reescrita neste mini-curso. Enfatizaremos a parte *equacional* de lógica de reescrita, ou seja, especificações em lógica equacional de pertinência (membership equational logic).

Maude

- Apresentaremos especificações em lógica equacional de pertinência “passo-a-passo”, utilizando a sintaxe da linguagem Maude.
- Maude é uma implementação de alta-performance para lógica de reescrita que poder ser obtida gratuitamente a partir do site `http://maude.cs.uiuc.edu`. Existem outras implementações de lógica de reescrita como ELAN, CafeObj e Obj. Cada uma delas possui características únicas mas Maude é sem dúvida a implementação com a melhor performance.

Especificações em lógica de pertinência

- Lógica equacional de pertinência é uma generalização de lógica equacional ordenada-sortida, em outras palavras, uma das lógicas que formalizam as especificações algébricas que vimos anteriormente.
- Primeiro apresentaremos especificações multi-sortidas (many-sorted equational specifications), passando a especificações ordenadas-sortidas (order-sorted equational specifications) para finalmente definir especificações de pertinência.

Especificações em lógica multi-sortida

```
fmod NAT is
 sort Nat .
 op 0 : -> Nat [ctor] .
 op s : Nat -> Nat [ctor] .
 op _+_ : Nat Nat -> Nat .

 vars M N : Nat .
 eq 0 + M = M .
 eq s(M) + N = s(M + N) .
endfm
```

Especificações em lógica multi-sortida

- Primeiro vamos entender a sintaxe desta especificação. Especificações em Maude são organizadas em módulos. No exemplo anterior o nome do módulo é `NAT`.
- O módulo `NAT` declara o (único) sort `Nat`, os construtores `0` e `s` e a operação `+`. Esse conjunto de declarações compõe a assinatura do módulo `NAT`.
- As equações em `NAT` especificam como se dá a soma de números naturais segundo esta notação, chamada de notação de Peano.

Especificações em lógica multi-sortida

```
                     \|||||||||||||||||/
                   --- Welcome to Maude ---
                     /|||||||||||||||||\
          Maude alpha 80 built: May  4 2003 23:12:17
               Copyright 1997-2003 SRI International
                     Mon May 26 17:47:33 2003
Maude> red s(0) + s(0) .
reduce in NAT : s(0) + s(0) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second
result Nat: s(s(0))
```

Especificações em lógica multi-sortida

- No exemplo anterior fizemos simplesmente a soma 1 + 1 na notação de Peano.
- Vejamos agora uma especificação para listas de naturais.

Especificações em lógica multi-sortida

```
fmod LIST-SIG is
 pr NAT .
 sort List .
 op nil : -> List [ctor] .
 op _,_ : List Nat -> List [ctor] .
 op length : List -> Nat .
 op concat : List List -> List .
 op first : List -> Nat .
endfm
Maude> red nil , s(0) , s(s(0)) .
result List: nil,s(0),s(s(0))
```

Especificações em lógica multi-sortida

- O termo `nil , s(0) , s(s(0))` representa uma lista com os números 1 e 2 e inicializada pela lista vazia `nil`.
- Vamos definir agora as equações para as outras operações que não são construtores.

Especificações em lógica multi-sortida

```
fmod LIST is
 inc LIST-SIG .
 vars N N’ : Nat . vars L L’ : List .
 eq length(nil) = 0 .
 eq length(L, N) = s(length(L)) .
 eq concat(L, nil) = L .
 eq concat(L, (L’ , N)) = concat(L, L’) , N .
endfm
Maude>
red concat((nil, s(0)), (nil, s(s(0)), s(s(s(0)))))
result List: nil,s(0),s(s(0)),s(s(s(0)))
```

Especificações em lógica multi-sortida

- As funções `length` e `concat` são definidas recursivamente. O caso base de `length`, dado pela lista vazia, resulta em `0`. O caso indutivo é dado pelo sucessor do comprimento do “resto” da lista.
- A função `concat` tem uma especificação similar.
- Vejamos agora especificações em lógica equacional ordenada-sortida.

Especificações em lógica ord.-sortida

- Especificações em lógica multi-sortida são desnecessariamente restritas na medida em que não podemos estabelecer *relações* entre os sorts.
- Por exemplo gostaríamos de relacionar os naturais aos inteiros, ou que pássaros são também animais.
- A especificação a seguir permite definirmos a função `div` como uma função total utilizando o sort `NzNat` para os naturais sem o zero. (É mostrado um trecho da especificação.)

Especificações em lógica ord.-sortida

```
fmod NAT-DIV is
   sorts Nat NzNat .
   subsort NzNat < Nat .
   op 0 : -> Nat . op s : Nat -> NzNat .
   ...
   op _-_ : Nat Nat -> Nat .
   op _<=_ : Nat Nat -> Bool .
   op _>_ : Nat Nat -> Bool .
   op _div_ : Nat NzNat -> Nat .
   ...
   ceq N div P = 0 if P > N .
   ceq N div P = s((N - P) div P) if P <= N .
endfm
```

Especificações equacionais

- Especificações equacionais devem possuir algumas propriedades:
  1. Terminação: derivações devem ser finitas.
  2. Unicidade dos resultados: não desejamos não-determinismo. Num sistema que termina desejamos que o resultado obtido seja único. (Especificações baseadas em regras não tem esta restrição em lógica de reescrita...)
  3. O resultado deve ser um termo produzido por um construtor.

Especificações equacionais

- Não iremos, neste mini-curso, formalizar estas propriedades nem discutir como verificá-las ou prová-las.
- Apesar de parecer óbvio que nossas especificações devam terminar, certos axiomas podem levar a não-terminação, como, por exemplo, a comutatividade.

```
eq A + B = B + A .
```

Especificações com atributos

- Maude permite que operadores sejam comutativos através da declaração de atributos que representam axiomas equacionais. Atualmente são suportados:
  - Associatividade, atributo `assoc`.
  - Comutatividade, atributo `comm`.
  - Identidade a esquerda, atributo `left id`, identidade a direita, atributo `right id`, ou identidade bi-lateral, atributo `id` com relação a um elemento de identidade.

Especificações com atributos

- A idéia básica é que a reescrita acontece nas *classes de equivalência* dos termos *modulo* os axiomas definidos nos atributos.
- Isso é atingido separando-se o conjunto de equações numa união disjunta formada pelas equações e os axiomas.
- Vejamos alguns exemplos.

Associatividade: Listas

```
fmod NAT-LIST-ASSOC is
   protecting NAT .
   sorts NeList List .
   subsorts Nat < NeList < List .
   op nil : -> List .
   op _ _ : List List -> List [assoc] .
   op _ _ : NeList NeList -> NeList [assoc] .
   op tail : NeList -> List .
   op head : NeList -> Nat .
   var N : Nat . var L : List .
   eq nil L = L . eq L nil = L .
   eq tail(N L) = L . eq head(N L) = N .
endfm
```

Associatividade: Listas

```
red tail( 1 2 3 4 ) .
result NeList: 2 3 4
```

- Note que não é necessário o uso de parênteses pois

  ```
  ( ( 1 2 ) 3 ) 4  = ( 1 ( 2 3 ) ) 4 =
  1 ( 2 ( 3 4 ) ) = ...
  ```

Associatividade + Identidade: Listas

```
fmod NAT-LIST-ASSOC-ID is
  ...
  op nil : -> List [ctor] .
  op _ _ : List List -> List [ctor assoc id: nil] .
  op _ _ : NeList NeList -> NeList
           [ctor assoc id: nil] .
  ...
endfm
```

Associatividade + Identidade: Listas

- É necessário cuidado com o uso da identidade no entanto.
- Uma operação como `op length : List -> Nat .` especificada pela equação `length(L L')` = `length(L)` + `length(L')` pode causar não-terminação:

```
length(L nil) = length(L) + length(nil) =
length(L nil) + length(nil) =
(length(L) + length(nil)) + length(nil) = ...
```

- Isso por causa da identidade `L = L nil`.

Assoc. + Com. + Id: Multisets

```
fmod NAT-MSET is
   protecting NAT .
   sorts NeMset Mset . subsort Nat < Mset .
   op empty-mset : -> Mset .
   op _ _ : Mset Mset -> Mset
            [assoc comm id: empty-mset] .
   op mult : Nat Mset -> Nat .
   ...
   vars N N' : Nat . var S : Mset .
   eq mult(N, empty-mset) = 0 .
   eq mult(N, N S) = s(0) + mult(N, S) .
   ceq mult(N, N' S) = mult(N, S) if N =/= N' .
   ...
```

Assoc. + Com. + Id: Multisets

- Um multiset ao contrário de um conjunto permite que um elemento apareça mais de uma vez no conjunto.
- Como não existe noção de ordem podemos utilizar o atributo para comutatividade.
- Assim como no exemplo das listas, uma equação como `size(S S')` = `size(S)` + `size(S')` poderia causar problemas de não terminação por causa do atributo identidade.

Especificações em lógica de pertinência

- Em especificações ordenadas-sortidas subsorts devem ser definidos por construtores, não sendo possível por exemplo definir um subsort de listas ordenadas definido por uma propriedade.
- Um outro problema mais sintático é exemplificado pelo termo `s(s(s(0))) div s(s(0)) - s(0)` (3 div (2 - 1)) que deveria ser igual a `s(s(s(0)))` (3) não chega nem a ser "parseado". Isso porque `s(s(0)) - s(0)` tem sort `Nat` e o operador `div` espera um `NzNat < Nat`.

Especificações em lógica de pertinência

- Lógica de pertinência resolve estes problemas introduzindo sorts como predicados e permitindo a definição de subsorts através de condições envolvendo equações e ou predicados de sort.
- Uma assinatura em lógica de pertinência é organizada não em termos de sorts mas em função dos *kinds*. Um kind é intuitivamente o componente conexo da relação de subsort.

Especificações em lógica de pertinência

- A notação ordenada-sortida é preservada por conveniência. A declaração

```
op _div_ :  Nat NzNat -> Nat .
```

corresponde a

```
op _div_ : [Nat] [NzNat] -> [Nat] .
cmb N div M : Nat if N : Nat and M : NzNat .
```

- Similarmente a declaração `NzNat < Nat`

corresponde a

```
cmb N : Nat if N : NzNat .
```

Especificações em lógica de pertinência

- Vamos ilustrar o uso de lógica de pertinência através da especificação a seguir que especifica um grafo e caminhos sobre um grafo.

Especificações em lógica de pertinência

```
fmod A-GRAPH is
   sorts Edge Node .
   ops n1 n2 n3 n4 n5 : -> Node .
   ops a b c d e f : -> Edge .
   ops source target : Edge -> Node .
   eq source(a) = n1 .  eq target(a) = n2 .
   eq source(b) = n1 .  eq target(b) = n3 .
   eq source(c) = n3 .  eq target(c) = n4 .
   eq source(d) = n4 .  eq target(d) = n2 .
   eq source(e) = n2 .  eq target(e) = n5 .
   eq source(f) = n2 .  eq target(f) = n1 .
endfm
```

Especificações em lógica de pertinência

```
fmod PATH is pr NAT . pr A-GRAPH .
   sorts Path Path? .
   subsorts Edge < Path < Path? .
   op _;_ : Path? Path? -> Path? [assoc] .
   ops source target : Path -> Node .
   op length : Path -> Nat .
   var E : Edge . var P : Path .
   cmb (E ; P) : Path if target(E) == source(P) .
   eq source(E ; P) = source(E) .
   eq target(P ; E) = target(E) .
   eq length(E) = s(0) .
   eq length(E ; P) = s(0) + length(P) .
endfm
```

Especificações em lógica de pertinência

```
red (b ; c ; d) .
result Path: b ; c ; d

red (a ; b ; c) .
result Path?: a ; b ; c

red source(a ; b ; c) .
result Error(Node): source(a ; b ; c)
```

Semântica de reescrita

- Vamos estudar agora uma especificação da semântica formal da linguagem bc em lógica equacional de pertinência, ou melhor, a semântica de reescrita de bc.
- Uma forma é mapear a especificação que já temos em semântica operacional para lógica equacional de pertinência.
- O mapeamento é bem simples: as regras de inferência (escritas em semântica natural) são mapeadas para equações condicionais em lógica equacional de pertinência.

Semântica de reescrita de bc

- A semântica de reescrita completa de bc, apesar de simples, ocuparia muitos slides. Vamos apresentar alguns trechos da especificação com o objetivo de ilustrar a técnica de especificação.
- A especificação completa pode ser obtida em `http://www.ic.uff.br/~cbraga/bc.maude`

Semântica de reescrita de bc

- Antes de analisarmos a especificação de expressões precisamos entender como dois conceitos utilizados em semântica operacional são representados em semântica de reescrita.
- O primeiro é a configuração. Em semântica de reescrita a configuração sendo reescrita é um par declarado com o operador

```
op <_,_> : Program Record -> ProgramState [ctor]
```

Semântica de reescrita de bc

- O segundo conceito é o encapsulamento dos componentes semânticos que em semântica operacional é representado através dos labels nas transições. Em semântica de reescrita este conceito é mapeado para o sort `Record`.
- O sort `Record` junto com a técnica de sorts abstratos que mostraremos mais a frente formam as soluções para o problema de modularidade em semântica de reescrita.

Semântica de reescrita de bc

- Expressões aritméticas

```
ceq < E1:Expr + E2:Expr , R:Record > =
    < V1:Value + V2:Value, R’’:Record >
 if < V1:Value , R’:Record > :=
    < E1:Expr , R:Record > /\
    < V2:Value , R’’:Record > :=
    < E2:Expr , R’:Record > .
```

Semântica de reescrita de bc

- O operador := é o operador de match.
- Ele produz uma substituição, ou seja, um conjunto de associações entre variáveis e valores a partir do casamento do padrão do lado esquerdo com o termo do lado direito.

Semântica de reescrita de bc

- Expressões aritméticas

```
ceq < E1:Expr / E2:Expr , R:Record > =
    < V1:Value / V2:NzValue , R''':Record >
 if < V1:Value , R':Record > :=
    < E1:Expr , R:Record > /\
    < V2:NzValue , R'':Record > :=
    < E2:Expr , R':Record > .
```

Semântica de reescrita de bc

- Essa equção ilustra a técnica de “non-zero sorts”. O sort `NzValue` representa os valores que são não-nulos e que portanto devem ser o tipo do segundo operando da operação `/`.
- As outras operações são similares a adição.
- Para especificar a avaliação de variáveis precisamos especificar o domínio semântico sobre o qual as variáveis serão avaliadas.

Semântica de reescrita de bc

- Memória abstrata

```
fmod MEMORY is
 pr RECORD . pr VAR . pr VALUE .
 sort Mem .
 subsort Mem < Component .
 op mem : -> Index .
 op lookup : Mem Var -> [Value] .
 op update : Mem Var Value -> Mem .
 mb mem <- M:Mem : Map .
endfm
```

Semântica de reescrita de bc

- Como dito anteriormente, junto com o sort `Record` a técnica de tipos abstratos de dados formam as soluções básicas para o problema de modularidade em semântica de reescrita.
- O problema que esta técnica se propõe a resolver são as chamadas *extensões não ortogonais*, ou seja, quando um componente semântico precisa ser estendido.

Semântica de reescrita de bc

- Note que a especificação de `Mem` tem uma semântica *loose*, ou seja, seu modelo não é o modelo inicial mas sim *todos* os possíveis modelos que a memória pode ter.
- Não definimos construtores para o sort `Mem`. Estes serão definidos pelas especializações da memória abstrata, ou seja, pelas memórias concretas.

Semântica de reescrita de bc

- Memória concreta.

```
fmod STORE is pr CELL . pr MEMORY .
 sort Store . subsorts Cell < Store < Mem .
 op empty-store : -> Store [ctor] .
 op __ : Store Store -> Store
          [ctor assoc comm id: empty-store] .
 eq lookup([V:Var,VL:Value] S:Store,V:Var) =
    VL:Value .
 eq lookup(S:Store,V:Var) = no-value [owise] .
```

Semântica de reescrita de bc

- Memória concreta.

```
 eq update([V:Var , VL:Value] S:Store ,
     V:Var, VL’:Value) =
    [V:Var , VL’:Value] S:Store .
 eq update(S:Store,V:Var,VL’:Value) =
    [V:Var , VL’:Value] S:Store [owise] .
endfm
```

Semântica de reescrita de bc

- A primeira especialização do sort `Mem` é o sort `Store`.
- O axioma para avaliação de variáveis é dado então, *de uma vez por todas* pela seguinte equação.

Semântica de reescrita de bc

- Avaliação de variáveis.

```
eq < SV:SimpleVar , {mem <- M:Mem , R2:PreRecord} >
   = < if lookup(M:Mem, SV:SimpleVar) == no-value
       then 0 else lookup(M:Mem, SV:SimpleVar) fi ,
       {mem <- M:Mem , R2:PreRecord} > .
```

Semântica de reescrita de bc

- Avaliação de atribuições.

```
ceq < SV:SimpleVar = E:Expr ,
      { mem <- M:Mem , R2:PreRecord } > =
      < noop ,
      { mem <- update(M':Mem, SV:SimpleVar,
                      V:Value) ,
      R2':PreRecord } >
   if < V:Value ,
      { mem <- M':Mem , R2':PreRecord } > :=
      < E:Expr ,
      { mem <- M:Mem , R2:PreRecord } > .
```

Semântica de reescrita de bc

- Assim como para avaliação de variáveis, o axioma para avaliação de atribuições é dado então, *de uma vez por todas*.
- Mesmo quando a memória for estendida com uma pilha para os registros de ativação, este axioma valerá pelo fato da equação estar especificada em termos da memória abstrata e não da memória concreta.

Semântica de reescrita de bc

- Avaliação de declaração de função.

```
op lambda(_;_;_) :
   SimpleVarList SimpleVarList Block -> Lambda .

eq < define F:FVar ( P:SimpleVarList )
     { auto A:SimpleVarList ; B:Block } ,
     { env <- E:Env , R2:PreRecord } > =
   < noop , { env <- override(E:Env, F:FVar,
      lambda(P:SimpleVarList ; A:SimpleVarList ;
             B:Block)),
     R2:PreRecord } > .
```

Semântica de reescrita de bc

- Assim como declaramos uma memória abstrata e a especializamos para uma memória concreta declaramos um ambiente abstrato e o especializamos para um ambiente para declaração de funções.
- O axioma para declaração de funções é também definido de uma vez por todas.

Semântica de reescrita de bc

- Memória composta pela memória global e a pilha.

```
fmod ESTORE is pr STORE . pr STACK .
 sort EStore . subsort EStore < Mem .
 op estore : Store Stack -> EStore [ctor] .
 op lookup : EStore Var -> Value .
 op update : EStore Var Value -> EStore .
 eq lookup(estore(S:Store, ST:Stack), V:Var) =
    if ( V:Var in top(ST:Stack) ) then
       lookupActRec(top(ST:Stack), V:Var)
    else lookup(S:Store, V:Var) fi .
```

Semântica de reescrita de bc

- Memória composta pela memória global e a pilha.

```
eq update(estore(S:Store, ST:Stack), V:Var,
    VL:Value) =
   if ( V:Var in top(ST:Stack) ) then
      estore(S:Store,
       (stack(updateActRec(top(ST:Stack),
         V:Var, VL:Value), pop(ST:Stack))))
   else
      estore(update(S:Store, V:Var, VL:Value),
       ST:Stack)
   fi .
```

Semântica de reescrita de bc

- Como vimos anteriormente a chamada de função utiliza também uma pilha.
- Por isso a memória abstrata deve mais uma vez ser especializada para este novo tipo de estrutura.
- Os próximos slides finalizam então a semântica de reescrita de bc.

Semântica de reescrita de bc

- Chamada de função.

```
ceq < F:FVar ( EL:ExprList ) , R:Record > =
    < apl(F:FVar, VL:ValueList ), R’:Record >
 if er(VL:ValueList, R’:Record) :=
    eval(EL:ExprList, R:Record) .
```

Semântica de reescrita de bc

```
ceq < apl(F:FVar, VL:ValueList ),
      { mem <- M:Mem ,
        env <- E:Env, R:PreRecord } > =
    < call(F:FVar, B:Block) ,
      { mem <- estore(S:Store,K':Stack),
        env <- E:Env, R:PreRecord } >
 if estore(S:Store,K:Stack) := M:Mem /\
    lambda(P:SimpleVarList ;
           A:SimpleVarList ; B:Block) :=
           find(E:Env, F:FVar) /\
    K':Stack :=
     stack(init-vals(P:SimpleVarList, VL:ValueList)
           init-val(A:SimpleVarList,0),K:Stack)
```

Semântica de reescrita de bc

- Chamada de função.

```
ceq < call(F:FVar, B:Block) ,
      { mem <- M:Mem , R:PreRecord } > =
    < C:Comp ,
      { mem <- estore(S:Store,pop(K:Stack)),
        R':PreRecord } >
```

Bibliografia

Este mini-curso foi desenvolvido baseado na seguinte literatura:

- Kenneth Slonneger e Barry L. Kurtz, Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach, Addison Wesley, 1995.
- Manuel Clavel et al, A Maude Tutorial, 2000. `http://maude.cs.uiuc.edu/tutorial`
- Peter Csaba Ølveczky, Formal Analysis of Distributed Systems in Maude-Part I, Konpendium INF 220, University of Oslo, 2003. `http://www.ifi.uio.no/inf220/`