MÁQUINAS DE ESTADO ABSTRATAS

Fabio Tirelo (DCC/UFMG)
Marcelo de Almeida Maia (DECOM/UFOP)
Vladimir Oliveira Di Iorio (DPI/UFV)
Roberto da Silva Bigonha (DCC/UFMG)

1999

INTRODUÇÃO

- ASM são utilizadas para modelagem matemática de sistemas dinâmicos discretos
- Idéia original: prover semântica operacional para algoritmos elaborando a tese de Turing
- Pode-se especificar um algoritmo em seu nível natural de abstração e com pouca codificação
- Conceitos utilizados são simples e bem conhecidos
- Leitura e escrita são fáceis
- Já foram aplicadas na especificação formal de Arquiteturas, Linguagens de Programação, Sistemas Distribuídos, Sistemas de Tempo Real, entre outros

SUMÁRIO

PRINCIPAIS CONCEITOS

- As ASM são máquinas abstratas, em que um estado é formado por funções e relações, definidas no *superuniverso*
- O conjunto dos nomes de funções e relações de um estado é o vocabulário do estado
- A interpretação de um nome de função ou relação é um mapeamento dos nomes pertencentes ao vocabulário nas respectivas funções ou relações
- A mudança de estado da máquina (transição) é dada por uma regra de transição
- Uma regra de transição modifica a interpretação de alguns nomes de função do vocabulário do estado

PRINCIPAIS CONCEITOS...

- Uma regra de transição de ASM tem a forma de um programa em uma linguagem imperativa
- A diferença principal é a ausência de iteração, pois este conceito está implícito na execução da máquina
- A execução da máquina consiste em executar a sua regra de transição repetidas vezes, modificando a cada vez o estado atual
- Assim, forma-se uma seqüência de estados de mesmo vocabulário e compostos por diferentes funções e relações
- Dizemos que a interpretação dos nomes pertencentes ao vocabulário é modificada de estado para estado na execução

VOCABULÁRIOS E ESTADOS

- Um *vocabulário* Υ é um conjunto finito de nomes de função e relação
- Exemplo:
  - $\{i, f\}$, onde i tem aridade 0 e f tem aridade 1
- Um *estado* de vocabulário Υ é um conjunto X junto com as interpretações dos nomes de Υ em X
- A interpretação de um nome de função r-ária é uma função $X^r \to X$
- A interpretação de um nome de relação r-ária é uma função $X^r \to \{true, false\}$

UNIVERSOS

- Se U é um nome de relação, então o conjunto

$$U = \{\bar{x} : U(\bar{x}) = true\}$$

é um universo em X

- Neste caso, dizemos que $U(\bar{x}) = true$ e $\bar{x} \in U$ são equivalentes
- Por exemplo, suponha que o nome de relação $Primos$ pertença ao vocabulário do estado
- Dizemos que o conjunto

$$Primos = \{\bar{x} : Primos(\bar{x}) = true\}$$

é um universo contido em X

REGRAS BÁSICAS

- As transições de estado da máquina são dadas por *regras de transição*
- Um novo estado é criado por meio da mudança na interpretação de alguns nomes de função ou relação
- As regras mais simples são *atualização*, *bloco* e *condicional*

REGRA DE ATUALIZAÇÃO

- A regra de atualização é uma regra da forma

$$f(t_1, \ldots, t_r) := t$$

onde f é um nome de função r-ária e $t, t_1, \ldots, t_r$ são termos

- Se f é um nome de relação, então t deve ser booleano
- No próximo estado, f é interpretado como uma função que, no ponto $t_1, \ldots, t_r$, tem o valor t
- Por exemplo, a regra

$$f(1) := 2$$

modifica a função f, tal que, no próximo estado, $f(1) = 2$

REGRA BLOCO

- A regra bloco é uma regra da forma

$$R_1, R_2$$

onde R_1 e R_2 são regras

- As regras R_1 e R_2 são executadas em paralelo
- Por exemplo, executando a regra

$$f(1) := 2, f(2) := 4$$

obtemos um estado, no qual $f(1) = 2$ e $f(2) = 4$

- Dizemos que uma regra bloco da forma

$$f(x) := y_1, f(x) := y_2$$

é *inconsistente*

REGRA CONDICIONAL

- A regra condicional é uma regra da forma

$$\textbf{if } g \textbf{ then } R_1 \textbf{ else } R_2 \textbf{ endif}$$

onde g é um termo booleano e R_1 e R_2 são regras

- Se g avaliar em $true$, R_1 é executada; senão, R_2 é executada
- Por exemplo, o estado criado pela regra

$$\textbf{if } f(1) = 2 \textbf{ then } f(2) := 4 \textbf{ else } f(2) := 5 \textbf{ endif}$$

cria um novo estado, onde a interpretação de f no ponto 2 pode ser 4 ou 5, dependendo do valor de $f(1)$

SEQÜÊNCIA DE REGRAS

- **Não existe** composição seqüencial de regras, isto é, uma regra da forma

$$R_1; R_2$$

onde primeiro executa-se R_1 e em seguida R_2

- Motivos:
  - Um programa em ASM descreve somente um passo do algoritmo
  - A composição seqüencial pode dificultar o raciocínio sobre a especificação

ESPECIFICAÇÃO ASM E EXECUÇÃO

- Uma especificação ASM contém a definição do estado inicial S_0 e da regra R
- A execução de uma especificação é uma seqüência de estados

$$\langle S_n : n \geq 0 \rangle$$

- Um estado S_{n+1} é obtido executando a regra R em S_n, para $n \geq 0$

EXEMPLO: FUNÇÃO FATORIAL

- O vocabulário é o conjunto $\Upsilon = \{i, fat\}$
- No estado inicial S_0, são dadas as seguintes interpretações
  - i é intepretado como 0
  - fat é interpretado como $\lambda x.x = 0 \rightarrow 1, undef$
- A regra da especificação é:

$$fat(i+1) := (i+1) \times fat(i), i := i+1$$

EXEMPLO: FUNÇÃO FATORIAL...

- Execução de $fat(i+1) := (i+1) \times fat(i), i := i+1$:

		$fat()$						
Estado	i	0	1	2	3	4	$\cdots$	n
S_0	0	1	*undef*	*undef*	*undef*	*undef*	$\cdots$	*undef*
S_1	1	1	1	*undef*	*undef*	*undef*	$\cdots$	*undef*
S_2	2	1	1	2	*undef*	*undef*	$\cdots$	*undef*
S_3	3	1	1	2	6	*undef*	$\cdots$	*undef*
S_4	4	1	1	2	6	24	$\cdots$	*undef*
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
S_n	n	1	1	2	6	24	$\cdots$	$n!$

- Pode-se ver facilmente que no estado $S_k, k \geq i$, $fat(i) = i!$

FUNÇÕES EXTERNAS

- Em geral, sistemas são afetados pelo ambiente
- Funções externas podem modelar interação com o ambiente
- Um exemplo de função externa é uma entrada fornecida pelo usuário
- Pode-se pensar em funções externas como *oráculos*
- A especificação fornece os argumentos e o oráculo fornece o resultado

REGRA *import*

- Uma regra *import* é da forma

$$\textbf{import } v \ R_0 \textbf{ endimport},$$

onde v é uma variável e R_0 é uma regra.

- Exemplo:

import v
  $Nodos(v) := true$
  $Esquerda(v) := undef$
  $Direita(v) := undef$
endimport

REGRA *choose*

- Uma regra *choose* é da forma

$$\textbf{choose } v \textbf{ in } U \textbf{ satisfying } g \ R_0 \textbf{ endchoose}$$

onde v é uma variável, U é o nome de um universo finito, g é um termo booleano e R_0 é uma regra

- Exemplo:

choose v **in** $Nodos$
  $Valor(v) := 1$
endchoose

REGRA *var*

- Uma regra *var* é da forma

$$\textbf{var } v \textbf{ ranges over } U \ R_0 \textbf{ endvar},$$

onde v é uma variável, U é o nome de um universo finito e R_0 é uma regra

- Exemplo:

var v **ranges over** $UmDoisTres$
  $f(v) := 0$
endvar

é equivalente a

$f(0) := 0$, $f(1) := 0$, $f(2) := 0$

EXEMPLO: PESQUISA BINÁRIA

- O problema é encontrar um valor k em um arranjo ordenado de inteiros, a, de comprimento n, indexado de 1 a n.
- Na solução proposta, o arranjo a será representado pela função f, de modo que $f(k) = a[k]$, para k inteiro.
- Interpretação dos nomes de função no estado inicial:
  - inf é interpretado como 1
  - sup é interpretado como n
  - k é interpretado como o valor da chave de pesquisa
  - o nome de relação $encontrado$ é interpretado como $false$
  - o nome de função f é interpretado como a função $\lambda k.1 \leq k \leq n \rightarrow a[k], undef$

EXEMPLO: PESQUISA BINÁRIA...

- A regra da especificação é:

if (**not** *encontrado* **and** $inf \leq sup$) **then**
  if $(k = f((inf + sup)/2))$ **then**
    $encontrado := true$,
    $pos := (inf + sup)/2$
  elseif $(k < f((inf + sup)/2))$ **then**
    $sup := (inf + sup)/2 - 1$
  else
    $inf := (inf + sup)/2 + 1$
  endif
endif

EXEMPLO: ORDENAÇÃO POR SELEÇÃO

- O problema é ordenar um arranjo de inteiros, a, de comprimento n, indexado de 1 a n
- Na solução proposta, o arranjo a será representado pela função f, de modo que $f(k) = a[k]$, para k inteiro
- Interpretação dos nomes de função no estado inicial:
  - o nome de função f é interpretado como a função $\lambda k.1 \leq k \leq n \rightarrow a[k], undef$

EXEMPLO: ORDENAÇÃO POR SELEÇÃO...

if $Modo = 1$ **and** $i < n$ **then**
  $k := i$, $j := i + 1$, $Modo := 2$
elseif $Modo = 2$ **then**
  if $j > n$ **then** $Modo := 3$
  elseif $f(j) < f(k)$ **then** $k := j$
  endif
  $j := j + 1$
elseif $Modo = 3$ **then**
  if $k \neq i$ **then**
    $f(k) := f(i)$, $f(i) := f(k)$
  endif
  $i := i + 1$, $Modo := 1$
endif

EXEMPLO: NÚMEROS PRIMOS

- O problema consistem em encontrar todos os primos menores ou iguais a um número n qualquer
- Definiremos um universo $Numeros$, subconjunto dos números inteiros, tal que $Numeros = \{x \in Inteiros : 2 \leq x \leq n\}$
- O vocabulário contém os nomes de função $primo$, de aridade 1, e x e n de aridade zero

EXEMPLO: NÚMEROS PRIMOS...

- No estado inicial as interpretações são dadas por:
  - x é interpretado como o valor 3
  - $primo$ aplicada a qualquer elemento pertencente ao universo $Numeros$ retorna $true$
- A regra marcará com $false$ todos os números de 2 a n que não forem primos
- No estado em que x for interpretado como $n+1$, a relação $primos$ mapeará os números primos em $true$ e os números compostos em $false$

EXEMPLO: NÚMEROS PRIMOS...

- A regra da especificação é a seguinte:

if $x \leq n$ **then**
  var y **ranges over** $Numeros$
    if $y < x$ **and** $x\%y = 0$ **then**
      $primo(x) := false$
    endif
  endvar
  $x := x + 1$
endif

EXEMPLO: SISTEMA OPERACIONAL

- O problema consiste em especificarmos o núcleo de um sistema operacional multiprogramado simples
- As operações especificadas serão:
  - criação de um novo processo
  - escalonamento de processos
  - difusão de uma mensagem para todos os processos ativos
  - recebimento de um sinal de interrupção

EXEMPLO: SISTEMA OPERACIONAL...

- O vocabulário contém os seguintes nomes de função e relação:
  - O universo $Processes$ que é o conjunto dos processos criados no sistema operacional
  - A função id, que associa cada processo à sua identificação
  - A função $messages$, que associa os processos às suas caixas de mensagens
  - A função $owner$, que associa cada recurso gerenciado ao processo que está utilizando
  - A relação $waiting$, que relaciona recursos aos processos que estão esperando para utilizá-lo

EXEMPLO: SISTEMA OPERACIONAL...

- Criação de um novo processo

import v
  $Processes(v) := true$,
  $id(v) := num\_procs + 1$,
  $num\_procs := num\_procs + 1$
endimport

- Escalonamento de processos

choose v **in** $Processes$ **satisfying** $waiting(v, recurso)$
  $owner(recurso) := id(v)$
endchoose

EXEMPLO: SISTEMA OPERACIONAL...

- Difusão de mensagem

var p **ranges over** $Processes$
  if $ativo(p)$ **then**
    $messages(id(p)) := append(messages(id(p)), mesg)$
  endif
endvar

- Recebimento de interrupção

if $interrupted$ **then**
  `...trata a interrupção...`
else
  `...continua realizando as operações normais...`
endif

O MODELO FORMAL DE ASM

- ASM são sistemas de transição que especificam computação
- Os estados de uma ASM são álgebras
- Os universos de álgebras que formam os estados da computação constituem o superuniverso da ASM

DEFINIÇÃO DE ÁLGEBRA

- Um sistema algébrico consiste em:
  1. Um conjunto não-vazio V, chamado o *domínio* da álgebra
  2. Um conjunto de *relações* sobre V
  3. Um conjunto de *operações* sobre V
  4. Um conjunto de *definições*
  5. Um conjunto de *axiomas*
  6. Um conjunto de *teoremas*

VOCABULÁRIO

- Um *vocabulário* Υ é uma coleção finita de nomes de função e relação
- Cada nome possui uma aridade fixa
- Estão presentes em todo vocabulário:
  - o sinal de igualdade
  - $true$, $false$ e $undef$
  - os operadores booleanos usuais

ESTADOS

- Um *estado* S é uma álgebra
- Um estado é uma tripla (Υ, X, Val), onde
  - Υ é um vocabulário
  - X é um conjunto denominado *superuniverso*
  - $Val : \Upsilon \rightarrow X^* \rightarrow X$ é a função de interpretação
- Para os conjuntos do superuniverso são definidas as operações usuais

FUNÇÕES ESTÁTICAS, DINÂMICAS E EXTERNAS

- Uma função é *dinâmica* se puder sofrer atualizações
- Caso contrário, dizemos que a função é *estática*
- São exemplos de funções estáticas:
  - Os operadores aritméticos
  - Os operadores booleanos
- São funções dinâmicas quaisquer funções que o usuário definir explicitamente como dinâmica
- Funções dinâmicas são "atualizadas" de estado para estado, dando o caráter dinâmico da especificação
- Funções externas

TERMOS

- *Termos* são definidos recursivamente como:
  - uma variável é um termo;
  - um nome de função ou relação de zero argumento é um termo;
  - se f é um nome de função ou relação r-ária, $r > 0$, e $\bar{x} = (x_1, \ldots, x_r)$, então $f(\bar{x})$ é um termo.
- Exemplos:
  - 1, 2, $true$, $undef$, v, i, $f(i)$, $g(i, f(v))$, etc.

TERMOS...

- Termos sem variáveis são interpretados, em um estado S, da seguinte maneira:
  - se f é um nome de função ou relação de zero argumento, sua interpretação é $Val_S(f)$;
  - se f é um nome de função r-ária e $(x_1, \ldots, x_r)$ é uma tupla de comprimento r, então

$$Val_S(f(x_1, \ldots, x_r)) = Val_S(f)(Val_S(x_1), \ldots, Val_S(x_r))$$

ENDEREÇOS

- Um *endereço* em um estado $S = (\Upsilon, X, Val)$ é um par $(f, \bar{x})$, onde
  - $f \in \Upsilon$ é um nome de função dinâmica
  - $\bar{x} \in X^*$ é uma tupla de comprimento igual à aridade de f
- Exemplos:
  - $(f, (1))$, $(g, (2, 3))$, etc.

ATUALIZAÇÕES

- Uma *atualização* em um estado $S = (\Upsilon, X, Val)$ é um par (l, y), onde
  - l é um endereço em S
  - $y \in X$
- A noção de atualização é importante para a definição de regras e disparo de regras
- Para cada regra definimos um conjunto de atualizações
- No exemplo anterior, são atualizações:
  - $(\underbrace{(f, (1))}_{\text{endereço}}, 5)$, $(\underbrace{(g, (2, 3))}_{\text{endereço}}, 4)$, etc.

CONSISTÊNCIA DE UM CONJUNTO DE ATUALIZAÇÕES

- O conjunto de atualizações Δ é *consistente* se não contiver atualizações diferentes para o mesmo endereço, isto é,

$$((f, \bar{x}), y_1) \in \Delta \wedge ((f, \bar{x}), y_2) \in \Delta \Rightarrow y_1 = y_2,$$

- Os conjuntos de atualização abaixo são consistentes:
  - $\{((f, (1)), 1), ((f, (2)), 1)\}$
  - $\{((f, (1)), 1), ((g, (1, 2)), 1)\}$
- Os conjuntos de atualização abaixo são inconsistentes:
  - $\{((f, (1)), 1), ((f, (1)), 2)\}$
  - $\{((f, (1)), 1), ((g, (1, 2)), 1), ((g, (1, 2)), 3)\}$

REGRAS

- Uma regra de atualização da forma $R \equiv f(\bar{t}) := t_0$ é uma regra
  - Seu conjunto de atualizações, em um estado S, é definido por

$$Updates(R, S) = \{((f, Val_S(\bar{t})), Val_S(t_0))\}$$

- O conjunto de atualizações da regra

$$f(1) := 2$$

é o conjunto

$$\{(f, (1)), 2)\}$$

REGRAS...

- um bloco de regras da forma $R \equiv R_1, \ldots, R_k$ é uma regra
  - Seu conjunto de atualizações é definido por

$$Updates\,(R, S) = \bigcup_{i=1}^{k} Updates\,(R_i, S)$$

  - Para disparar um bloco de regras, disparam-se todas as regras que o compõem *simultaneamente*
- O conjunto de atualizações da regra

$$f(1) := 2, g(1, 2) := 3$$

é o conjunto

$$\{((f, (1)), 2), ((g, (1, 2)), 3)\}$$

REGRAS...

- se k é natural, $g_0, \ldots, g_k$ são termos booleanos e $R_0, \ldots, R_k$ são regras, então

$$\begin{aligned} R \equiv\ & \textbf{if } g_0 \textbf{ then } R_0 \\ & \textbf{elseif } g_1 \textbf{ then } R_1 \\ & \ldots \\ & \textbf{elseif } g_k \textbf{ then } R_k \\ & \textbf{endif} \end{aligned}$$

é uma regra

  - Seu conjunto de atualizações é definido por

$$Updates(R, S) = \begin{cases} Updates(R_i, S) & \text{se } g_i \wedge \forall j < i : \neg g_j \\ \emptyset & \text{se } \forall i : \neg g_i \end{cases}$$

DISPARO DE UMA REGRA

- O *disparo de uma regra* R em um estado S produz um novo estado S', tal que, se $Updates(R, S)$ for consistente, então

$$Val_{S'}(f, \bar{x}) = \begin{cases} y & \text{se } ((f, \bar{x}), y) \in Updates(R, S) \\ Val_S(f, \bar{x}) & \text{caso contrário.} \end{cases}$$

- Se o conjunto $Updates(R, S)$ não for consistente, então o efeito do disparo de R em S será nulo
- Outra abordagem possível é fazer uma escolha não-determinista da atualização que será disparada
- Definiremos $Fire(R, S)$ como o estado resultante do disparo de R em S

ESPECIFICAÇÃO ASM E EXECUÇÃO

- Uma *especificação ASM* é uma tupla da forma $(\Upsilon, \mathcal{A}, S_0, \mathcal{P})$, onde
  - Υ é um vocabulário
  - $\mathcal{A}$ é um conjunto de Υ-álgebras ou estados S
  - $S_0 \in \mathcal{A}$ é o estado inicial
  - $\mathcal{P}$ é uma regra
- A *execução de uma especificação ASM* $(\Upsilon, \mathcal{A}, S_0, \mathcal{P})$ é uma seqüência $\mathcal{S} = \langle S_n : n \geq 0 \rangle$ de estados pertencentes a $\mathcal{A}$, onde $S_{n+1} = Fire(\mathcal{P}, S_n)$

EXEMPLO DE ESPECIFICAÇÃO FORMAL

- Uma especificação ASM para a função fatorial é $ASM_{fat} = (\Upsilon, \mathcal{A}, S_0, P_{fat})$, onde:
  - $\Upsilon = \Upsilon_0 \cup \Upsilon_e$, onde Υ_0 é um conjunto de símbolos pré-definidos e $\Upsilon_e = \{fat, i\}$;
  - $\mathcal{A}$ é o conjunto das Υ-álgebras que modelam os estados;
  - a interpretação dos nomes de função pertencentes a Υ em $S_0 \in \mathcal{A}$ é dado pela função Val_{S_0}, tal que
$$\begin{aligned} Val_{S_0}(i) &= 0 \\ Val_{S_0}(fat) &= \lambda n.n = 0 \rightarrow 1, undef \end{aligned}$$
  - P_{fat} é dado pela regra:
$$fat(i+1) := (i+1) \times fat(i), i := i+1$$

UMA PROVA DE CORREÇÃO DA ESPECIFICAÇÃO

- A execução de ASM_{fat} é a seqüência $\mathcal{S}_{fat} = \langle S_0, S_1, \cdots \rangle$ de elementos de $\mathcal{A}$, onde cada $S_{n+1} = Fire(P_{fat}, S_n)$, para $n \geq 0$
- Provaremos que em S_n, $fat(n) = n!$
- Para $i = 0$, $Val_{S_i}(i) = Val_{S_0}(i) = 0$ e $Val_{S_i}(fat)(Val_{S_i}(i)) = 1 = 0!$
- Supondo que, no estado S_i, $Val_{S_i}(fat)(Val_{S_i}(i)) = (Val_{S_i}(i))!$, temos que, no estado S_{i+1}, $Val_{S_{i+1}}(i) = Val_{S_i}(i) + 1$ e
$$\begin{aligned} Val_{S_{i+1}}(fat)(Val_{S_{i+1}}(i)) &= (Val_{S_i}(i)+1) \times Val_{S_i}(fat)(Val_{S_i}(i)) \\ &= (Val_{S_i}(i)+1) \times (Val_{S_i}(i))! \\ &= (Val_{S_i}(i)+1)! \\ &= (Val_{S_{i+1}}(i))! \end{aligned}$$

REGRA *import*

- A regra *import* é uma regra da forma
$$R \equiv \textbf{import } v \ R_0 \textbf{ endimport},$$
onde v é uma variável e R_0 é uma regra
- A semântica desta regra é:
  - Escolhe-se um elemento a do universo $Reserve$ e associa-o à variável v
  - Executa-se a regra R_0, no estado S_a
  - O conjunto de atualizações para esta regra é dado por
$$Updates(R, S) = \{((Reserve, a), false)\} \cup Updates(R_0, S_a)$$

REGRA *var*

- A regra *var* é uma regra da forma

$$R \equiv \textbf{var } v \textbf{ ranges over } U \ R_0 \textbf{ endvar}$$

- A semântica desta regra é:
  - Para cada elemento x de U, executamos a regra R_0 no estado S_x
  - Esta execução cria o conjunto de atualizações $Updates(R_0, S_x)$
  - O efeito da regra é a união de todos os conjuntos $Updates(R_0, S_x)$, tal que $x \in U$, isto é,

$$Updates(R, S) = \bigcup_{x \in U} Updates(R_0, S_x)$$

REGRA *choose*

- A regra *choose* é uma regra da forma

$$\textbf{choose } v \textbf{ in } U \textbf{ satisfying } g \ R_0 \textbf{ endchoose},$$

  onde v é uma variável, U é o nome de um universo finito, g é um termo booleano e R_0 é uma regra
- O efeito desta regra é executar a regra R_0 no estado S_a
- O elemento a é escolhido de maneira não-determinista e deve tornar a guarda g verdadeira.

A LINGUAGEM *Tiny*

- As ASM já foram utilizadas para definição formal de diversas linguagens de programação: Java, C, Prolog, entre outras
- As ASM permitem que se especifique, de maneira bastante simples, a semântica operacional de uma linguagem de programação
- A linguagem *Tiny* é uma linguagem imperativa de pequeno porte, que contém somente comandos e expressões

SINTAXE E SEMÂNTICA DE *Tiny*

- A sintaxe abstrata de *Tiny* é dada por

$$\begin{aligned} E &::= 0 \mid 1 \mid true \mid false \mid read \mid I \\ &\quad \mid not\ E \mid E_1 = E_2 \mid E_1 + E_2 \mid (E) \\ C &::= I := E \mid output\ E \mid if\ E\ then\ C_1\ else\ C_2 \\ &\quad \mid while\ E\ do\ C \mid C_1; C_2 \mid (C) \end{aligned}$$

- Todo comando de *Tiny*, quando executado, modifica um estado composto por três elementos:
  - a memória
  - a entrada
  - a saída

REPRESENTAÇÃO DE PROGRAMAS *Tiny*

Expressões e Comandos de *Tiny*	**Representação**
0	$[0]$
1	$[1]$
$false$	[FALSE]
$true$	[TRUE]
$read$	[READ]
I	$[I]$
$not\ E$	[NOT, E]
$E_1 = E_2$	[EQUALS, E_1, E_2]
$E_1 + E_2$	[ADD, E_1, E_2]
$I := E$	[ASSIGN, I, E]
$output\ E$	[OUTPUT, E]
$if\ E\ then\ C_1\ else\ C_2$	[IF, E, C_1, C_2]
$while\ E\ do\ C$	[WHILE, E, C]
$C_1; C_2$	$[C_1, C_2]$

O MODELO OPERACIONAL

- A Semântica Operacional descreve a semântica de uma construção, mostrando como ela é executada em uma máquina abstrata
- Componentes da arquitetura (o vocabulário Υ):
  - $program$ – inicialmente é o programa fonte
  - $memory$ – inicialmente é $\lambda x.undef$
  - $input$ – inicialmente é a entrada
  - $output$ – inicialmente igual a $[\,]$
  - $opstack$ – inicialmente igual a $[\,]$
- Utilizaremos ASM para descrever a execução de um programa nesta máquina abstrata

DEFINIÇÃO ASM DE *Tiny*

- A regra da especificação é

  if $program \neq [\,]$ **and not** $error$ **then**
  `...Passo da Interpretação...`
  endif

- Temos a seguinte semântica: executa-se um "passo da interpretação", até que todo o programa tenha sido executado ou tenha ocorrido um erro
- Um passo da interpretação é uma lista de regras da forma:

  ...
  if $Head(program) =$ IF **then** $\cdots$ **endif,**
  if $Head(program) =$ WHILE **then** $\cdots$ **endif,**
  ...

DEFINIÇÃO ASM DE *Tiny*...

- Um passo da interpretação consistem em
  - consultar o primeiro elemento da lista $program$
  - se este elemento for uma lista, promover a sua expansão
  - se for um identificador, substitui-lo por seu valor na memória
  - se for alguma palavra reservada da interpretação, executar as ações necessárias; tais ações podem alterar a pilha $opstack$
  - se for um número inteiro ou um valor booleano, consultar a pilha $opstack$ para determinar a ação que será executada

DEFINIÇÃO ASM DE *Tiny*...

- Regra para expansão da lista

if $List(hd(program))$ **then**
  $program := concat(hd(program), tl(program))$
endif

$$program = [[\mathsf{ADD}, E_1, E_2], \cdots]$$
$$\Downarrow$$
$$program = [\mathsf{ADD}, E_1, E_2, \cdots]$$

DEFINIÇÃO ASM DE *Tiny*...

- Regra para a expressão de leitura

if $hd(program) = \mathsf{READ}$ **then**
  if $input = [\,]$ **then** $error := true$
  else
    $program := cons(hd(input), tl(program))$,
    $input := tl(input)$
  endif
endif

$$program = [\mathsf{READ}, P \cdots]$$
$$input = [1, 2, I \cdots]$$
$$\Downarrow$$
$$program = [1, P \cdots]$$
$$input = [2, I \cdots]$$

DEFINIÇÃO ASM DE *Tiny*...

- Regra para a expressão identificador

if $String(hd(program))$ **then**
  if $memory(hd(program)) = undef$ **then**
    $error := true$
  else
    $program := cons(memory(hd(program)), tl(program))$
  endif
endif

$$program = [\text{“}x\text{”}, P \cdots]$$
$$memory(\text{“}x\text{”}) = 10$$
$$\Downarrow$$
$$program = [10, P \cdots]$$

DEFINIÇÃO ASM DE *Tiny*...

- Regra para a expressão de negação lógica

if $hd(program) = \mathsf{NOT}$ **then**
  $program := tl(program)$,
  $opstack := cons(\mathsf{NOT}, opstack)$
endif

$$program = [\mathsf{NOT}, E_1, P \cdots]$$
$$opstack = [O \cdots]$$
$$\Downarrow$$
$$program = [E_1, P \cdots]$$
$$opstack = [\mathsf{NOT}, O \cdots]$$

DEFINIÇÃO ASM DE *Tiny*...

- Regra para a expressão de igualdade

if $hd(program) =$ EQUALS **then**
$program := tl(program)$,
$opstack := cons(\text{EXCHANGE}, cons(\text{EQUALS}, opstack))$
endif

$$\begin{array}{|rcl|}\hline program &=& [\text{EQUALS}, E_1, E_2, P \cdots] \\ opstack &=& [O \cdots] \\ \hline \end{array}$$

$$\Downarrow$$

$$\begin{array}{|rcl|}\hline program &=& [E_1, E_2, P \cdots] \\ opstack &=& [\text{EXCHANGE}, \text{EQUALS}, O \cdots] \\ \hline \end{array}$$

DEFINIÇÃO ASM DE *Tiny*...

- Regra para a expressão de adição

if $hd(program) =$ ADD **then**
$program := tl(program)$,
$opstack := cons(\text{EXCHANGE}, cons(\text{ADD}, opstack))$
endif

$$\begin{array}{|rcl|}\hline program &=& [\text{ADD}, E_1, E_2, P \cdots] \\ opstack &=& [O \cdots] \\ \hline \end{array}$$

$$\Downarrow$$

$$\begin{array}{|rcl|}\hline program &=& [E_1, E_2, P \cdots] \\ opstack &=& [\text{EXCHANGE}, \text{ADD}, O \cdots] \\ \hline \end{array}$$

DEFINIÇÃO ASM DE *Tiny*...

- Regra para o comando de atribuição

if $hd(program) =$ ASSIGN **then**
$program := tl(tl(program))$,
$opstack := cons(\text{ASSIGN}, cons(hd(tl(program)), opstack))$
endif

$$\begin{array}{|rcl|}\hline program &=& [\text{ASSIGN}, \text{“}x\text{”}, E_1, P \cdots] \\ opstack &=& [O \cdots] \\ \hline \end{array}$$

$$\Downarrow$$

$$\begin{array}{|rcl|}\hline program &=& [E_1, P \cdots] \\ opstack &=& [\text{ASSIGN}, \text{“}x\text{”}, O \cdots] \\ \hline \end{array}$$

DEFINIÇÃO ASM DE *Tiny*...

- Regra para o comando de impressão

if $hd(program) =$ OUTPUT **then**
$program := tl(program)$,
$opstack := cons(\text{OUTPUT}, opstack)$
endif

$$\begin{array}{|rcl|}\hline program &=& [\text{OUTPUT}, E_1, P \cdots] \\ opstack &=& [O \cdots] \\ \hline \end{array}$$

$$\Downarrow$$

$$\begin{array}{|rcl|}\hline program &=& [E_1, P \cdots] \\ opstack &=& [\text{OUTPUT}, O \cdots] \\ \hline \end{array}$$

DEFINIÇÃO ASM DE *Tiny*...

- Regra para o comando de condicional

if $hd(program) =$ IF **then**
$program := tl(program)$,
$opstack := cons(\text{COND}, opstack)$
endif

$$\begin{array}{|rcl|}\hline program &=& [\text{IF}, E, C_1, C_2, P \cdots] \\ opstack &=& [O \cdots] \\ \hline \end{array}$$
$$\Downarrow$$
$$\begin{array}{|rcl|}\hline program &=& [E, C_1, C_2, P \cdots] \\ opstack &=& [\text{COND}, O \cdots] \\ \hline \end{array}$$

DEFINIÇÃO ASM DE *Tiny*...

- Regra para o comando de repetição

if $hd(program) =$ WHILE **then**
$opstack := cons(\text{COND}, opstack)$,
$program := cons(\ WhileExp(program), cons(\ WhileTrue(program),$
$cons([\ ], WhileCont(program))))$
endif

$$\begin{array}{|rcl|}\hline program &=& [\text{WHILE}, E, C, P \cdots] \\ opstack &=& [O \cdots] \\ \hline \end{array}$$
$$\Downarrow$$
$$\begin{array}{|rcl|}\hline program &=& [E, [C, [\text{WHILE}, E, C]], [\ ], P \cdots] \\ opstack &=& [\text{COND}, O \cdots] \\ \hline \end{array}$$

DEFINIÇÃO ASM DE *Tiny*...

- A lista $opstack$ é consultada se o elemento no início de $program$ for um número inteiro ou um valor booleano
- Impressão de um valor:

$$\begin{array}{|rcl|}\hline program &=& [10, P \cdots] \\ opstack &=& [\text{OUTPUT}, X \cdots] \\ output &=& [O \cdots] \\ \hline \end{array}$$
$$\Downarrow$$
$$\begin{array}{|rcl|}\hline program &=& [P \cdots] \\ opstack &=& [X \cdots] \\ output &=& [10, O \cdots] \\ \hline \end{array}$$

DEFINIÇÃO ASM DE *Tiny*...

- Atribuição de um valor a uma variável:

$$\begin{array}{|rcl|}\hline program &=& [10, P \cdots] \\ opstack &=& [\text{ASSIGN}, \text{“}x\text{”}, O \cdots] \\ \hline \end{array}$$
$$\Downarrow$$
$$\begin{array}{|rcl|}\hline program &=& [P \cdots] \\ opstack &=& [O \cdots] \\ memory(\text{“}x\text{”}) &=& 10 \\ \hline \end{array}$$

- Avaliação da segunda expressão de uma operação binária:

$$\begin{array}{|rcl|}\hline program &=& [10, E_2, P \cdots] \\ opstack &=& [\text{EXCHANGE}, O \cdots] \\ \hline \end{array}$$
$$\Downarrow$$
$$\begin{array}{|rcl|}\hline program &=& [E_2, 10, P \cdots] \\ opstack &=& [O \cdots] \\ \hline \end{array}$$

DEFINIÇÃO ASM DE *Tiny*...

- Adição de valores:

$$\begin{array}{r} program = [10, 25, P \cdots] \\ opstack = [\mathsf{ADD}, O \cdots] \end{array}$$

$$\Downarrow$$

$$\begin{array}{r} program = [35, P \cdots] \\ opstack = [O \cdots] \end{array}$$

- Negação lógica:

$$\begin{array}{r} program = [\mathsf{TRUE}, P \cdots] \\ output = [\mathsf{NOT}, O \cdots] \end{array}$$

$$\Downarrow$$

$$\begin{array}{r} program = [\mathsf{FALSE}, P \cdots] \\ output = [O \cdots] \end{array}$$

DEFINIÇÃO ASM DE *Tiny*...

- Decisão de um condicional:

$$\begin{array}{r} program = [\mathsf{TRUE}, C_1, C_2, P \cdots] \\ opstack = [\mathsf{COND}, O \cdots] \end{array}$$

$$\Downarrow$$

$$\begin{array}{r} program = [C_1, P \cdots] \\ opstack = [O \cdots] \end{array}$$

- Expressão de comparação:

$$\begin{array}{r} program = [10, 25, P \cdots] \\ output = [\mathsf{EQUALS}, O \cdots] \end{array}$$

$$\Downarrow$$

$$\begin{array}{r} program = [\mathsf{FALSE}, P \cdots] \\ output = [O \cdots] \end{array}$$

ASM MULTIAGENTES

- Uma ASM Multiagente consiste em:
  - Conjunto finito de programas (*Módulos*)
  - A cada módulo está associado um conjunto finito de *agentes*
  - O vocabulário Υ é comum a todos os agentes
  - Em um agente a, $Self \in \Upsilon$ é interpretado como a

EXECUÇÃO PARCIALMENTE ORDENADA

- Uma *execução parcialmente ordenada* é uma tripla (M, A, σ) que satisfaz:
  - M é um conjunto parcialmente ordenado dos movimentos realizados pelos vários agentes
  - A é uma função sobre M tal que todo conjunto não-vazio $\{x : A(x) = a\}$ é linearmente ordenado
  - σ é uma função que associa um estado a cada *segmento inicial* de M
  - Condição de coerência

DINING PHILOSOPHERS

- Initial Values:

var q **ranges over** *Philosophers*
  $Status(q) :=$ "*thinking*",
  $Holder(q) :=$ *undef*
endvar

- Transition Rules for philosopher p:

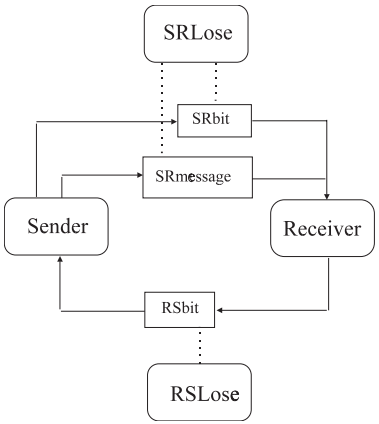
if $Status(p) =$ "*thinking*" **then**
  $Status(LeftFork(p)) :=$ "*hungry*"
  $Status(RightFork(p)) :=$ "*hungry*"
endif

DINING PHILOSOPHERS...

- Transition Rules (cont.):

if $Status(p) =$ "*hungry*" **and** $Holder(LeftFork(p)) =$ *undef*
  and $Holder(RightFork(p)) =$ *undef* **then**
  $Holder(LeftFork(p)) := p$,
  $Holder(RightFork(p)) := p$,
  $Status(p) :=$ "*eating*"
endif
if $Status(p) =$ "*eating*" **then**
  $Holder(LeftFork(p)) :=$ *undef*,
  $Holder(RightFork(p)) :=$ *undef*,
  $Status(p) :=$ "*thinking*"
endif

ALTERNATING BIT PROTOCOL



ALTERNATING BIT PROTOCOL

- Rules for **Sender**:

if $Clock > Slast + timeout$ **then**
  $SRmessage := Smessage$,
  $SRbit := Sbit$,
  $Slast := Clock$
endif
if $RSbit = Sbit$ **then**
  $Scont := Scont + 1$,
  $Smessage := InputFile(Scont + 1)$,
  $Sbit := \neg Sbit$
endif

ALTERNATING BIT PROTOCOL

- Rules for **Receiver**:

if $Clock > Rlast + timeout$ **then**
$RSbit := Rbit$,
$Rlast := Clock$
endif
if $SRmessage \neq undef$ **and** $SRbit \neq Rbit$ **then**
$Rcont := Rcont + 1$,
$OutputFile(Rcont + 1) := SRmessage$,
$Rbit := \neg Rbit$
endif

ALTERNATING BIT PROTOCOL

- Rules for **SRLose**:

$SRbit := undef$,
$SRmessage := undef$

- Rules for **RSLose**:

$RSbit := undef$

AVALIAÇÃO DA METODOLOGIA

1. Precisão
2. Demonstração de correção
3. Generalidade
4. Facilidade de Aprendizado
5. Facilidade de Leitura e de Escrita
6. Escalabilidade
7. Possibilidade de Execução

CONSIDERAÇÕES FINAIS

- Especificação de Linguagens ou Sistemas
- Trabalhos de Pesquisa
  - Interpretadores
  - Ferramentas de auxílio
    - Provadores de teoremas,
    - Verificadores de tipos
    - Módulos
- http://www.eecs.umich.edu/gasm
- http://www.uni-paderborn.edu/cs/asm.html