Biblioteca para Tipos Abstratos de Dados em *Machĭna*

Letícia Decker de Sousa

Profa. Mariza Andrade da Silva Bigonha
Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Linguagens de Programação

26 de julho de 2008

Proposta

Esse projeto está inserido dentro de um projeto de escopo maior que é o Projeto *Machĭna*, que inclui a construção da especificação da linguagem, do compilador, inserção de orientação a aspecto à linguagem e agora, construção de uma biblioteca de *TADs*.

Objetivos

- Transformar esses *TADs* em bibliotecas de *Machĭna*, com o intuito de facilitar implementações futuras nessa linguagem.

- Reestruturar tipos abstratos de dados (*TADs*) implementados em *Machĭna*, adaptando-os às modicações sofridas pela linguagem ao longo da sua evolução.

ASM - Noção Teórica

- Idéia original: *todo algoritmo é simulado por uma mquina de Turing apropriada*.
- Inconviniente: pode ser necessária uma longa sequência de passos na máquina para simular um único passo.
- Solução: *ASM*, pois é mais natural e mais próxima.

O Modelo Formal de *ASM*

ASM são sistemas de transição que especificam computação cujos estados são álgebras.

Os universos de álgebra que formam os estados da computação constituem o superuniverso da *ASM*.

Provê recursos expressivos para especificar a semântica operacional de sistemas dinâmicos discretos, em um nível de abstração natural e de maneira direta, livre de codificação.

Diminuir a distância entre modelos formais e práticos de especificação.

Máquinas de estados que simulem passo a passo um algoritmo.

Vantagens de *ASM*

- Possibilidade de de se executar uma especificação, facilitando a tarefa de encontrar erros.
- O modelo possui o recurso de modelar concorrência e não-determinismo.
- A existência de uma teoria matemática subjacente, a teoria de *Álgebra Evolutiva*, que permite a prova de propriedades da especificação.
- Modelo de linguagens imperativas não possuem tal característica.

... Vantagens de *ASM*

- Possibilidade de de se executar uma especificação, facilitando a tarefa de encontrar erros.
- O modelo possui o recurso de modelar concorrência e não-determinismo.
- A existência de uma teoria matemática subjacente, a teoria de Álgebra Evolutiva, que permite a prova de propriedades da especificação.
- Modelo de linguagens imperativas não possuem tal característica.

Álgebra Evolutiva

- *ASM* são máquinas de estado abstratas, em que um estado formado por funções e relações.
- Funções e relações possuem nomes, definidos no superuniverso do estado.
- Conjunto de nome de função ou relação de um estado o vocabulário do estado.

... Álgebra Evolutiva

- A regra de transição de *ASM* tem a aparência de um programa, excluindo-se o conceito de iteração, pois este está implícito na execução da máquina.
- Novo estado é criado a partir do estado atual executando-se uma regra de transição, por meio da mudança de interpretação de cada nome de função.
- Regras mais simples: condicional, bloco e atualização.

A Linguagem *Machĭna*

Linguagem *Machĭna* é baseada em *ASM* e oferece as seguintes facilidades:

- Estruturas para modularização e mecanismos de visibilidade e proteção;
- Extensibilidade de tipos;
- Sequenciadores de regras;
- Sistema fortemente tipado, com rico conjunto de tipos pré-definidos;

Características de *Machĭna*

- Invariante para a execução da regra de transição da máquina abstrata;
- Regras de transição de estado;
- Multiagentes, com capacidade de autonomia, independência, consciência de contexto e sociabilidade, introduzida na linguagem de maneira simples e direta;
- Abstração de regras de transição, incluindo ações e iterações, que podem por exemplo, ser executadas a partir de outra máquina, criado a noção de submáquinas.

Módulo em *Machĭna*

```
Modulo nome-do-modulo
      import: elementos importados
      include: elementos incluido
      algebra:
            elementos declarados (funcoes e tipos)
      abstractions:
            declaracao de abstracoes (acoes e iteracoes)
      initial state:
            inicializacoes de funcoes dinamicas
      transition:
            regra de transicao de estado
      invariant:
            invariante da execucao
end nome-do-modulo
```

Módulo de Programas

- *import*: coloca no escopo do módulo a interface dos agentes com os quais o agente do módulo deve se comunicar. A interface dos agentes contém as assinaturas das abstrações de regras.

- *include*: define os módulos secundários e seus elementos. Também serve para controle de visibiliade de elementos declarados públicos ou compartilhados em um módulo.

... Módulo de Programas

- *algebra*: define os elementos da álgebra subjacente ao modelo (tipos e funções).
- *initial state*: serve para inicializar de funções dinâmicas.
- *abstractions*: define as abstrações de regras de transição, que podem ser usadas localmente ou exportadas.

- *transition*: define a regra de transição de estado do agente, a qual é executada repetidamente quando o agente é disparado.

- *invariant*: define a condição envolvendo os elementos de um módulo que deve ser invariante durante sua execução.

Durante duas iterações da regra de transição do módulo, o invariante é verificado pelo sistema de execução de *Machĭna*.

Sistema de Tipos

- *Tipos Básicos*: *Bool, Char, Int, Real, String* e *Promise*.
- *Tipos Compostos*: listas, agentes, tuplas, uniões disjuntas, enumeraçóes, funcionais e tipos definidos pelo programador.
- *Tipos Genéricos*: listas genéricas (*List*), conjuntos genéricos (*Set*), agentes genéricos (*Agent*), tipo *?* é a união disjunta de todos os tipos.
- *Tipos Arquivos*: arquivo stream (*Input* e *Output*) e aquivos binários (file of T).

Regras de Transição de Estado

Normalmente, os passos são paralelizados, para a seqencialização dos passos usa-se a palavra-chave *step*.

- *Atualização*: formada por um identificador, que deve ser o nome de uma função declarada como dinâmica, seguido por seus argumentos, os quais são usados para a determinação do endereço de atualização, e uma expressão, cujo o valor é atribuído ao endereço formado.

... Regras de Transição de Estado

- *Bloco de Regra*: composta por uma sequência de regras, separadas por ponto-e-vírgula (;), que devem ser executadas simultaneamente.

- *if* : tem o efeito de executar uma regra escolhida a partir do valor de uma guarda.

```
if g1 then R1
     elseif g2 then R2
     elseif gK then Rk
     else Rk+1
end
```

... Regras de Transição de Estado

- *Case*: tem o efeito de executar uma regra escolhida a partir dos valores de guardas de diversar alternativas possíveis.

```
case head(s) * head(tail(s))
     of 1 $=>$ x:= 11
     of 3 $=>$ x:= 17
     of 4 $=>$ x:= 19
     otherwise $=>$ x:= 23
end
```

... Regras de Transição de Estado

- *with* : forma especial de *Case*.

```
with e
     as x1: T1 $=>$ R1(x1);
     as [a b c ...] $=>$ R2(a, b, c);
     as a::c $=>$ R3(a, c);
     as (a, b, c,...) $=>$ R4(a, c);
     otherwise $=>$ R5
end
```

- *choose*: escolher não-deterministicamente elementos dos domínios dados que satisfaçam a guarda e executar a regra dada, associando os nomes de identificadores aos elementos escolhidos.

... Regras de Transição de Estado

- *Chamadas de Abstração*: gera um conjunto de atualizações, que é incluído no conjunto de atualizações existentes no ponto de chamada.

Expressões

Operadores (em ordem de precedência):

- old
- ::
- + - not
- * / and
- + - or xor
- ..
- = ! = < > <= >= is

Define-se tipo abstrato de dado (*TAD*) como uma estrutura de um programa que satisfaz às seguintes condições:

- É caracterizados por um conjunto bem definidos de operações.
- É uma encapsulação que define a representação dos valores do tipo sobre os quais as operações descritas dentro da referida encapsulação atuam.
- Existe um controle de visibilidade: a estrutura interna do tipo não é visível ao usuário, que só poderá acessá-la por meio das operações que forem descritas internamente ao módulo. Mas a interface do tipo é público.
- Operações e definição de tipo são descritas dentro de uma única unidade sintática, sendo que outras unidades de programa tem permissão de criar variáveis do tipo citado.

TADs em *Machĭna*

- Para a implementação de um tipo abstrato em *Machĭna* é necessário concentrar em um único módulo: o tipo que se deseja manipular, torná-lo público via a palavra-chave public imediatamente anterior à palavra-chave type, e posteriormente, denomina-se o tipo seguido por "=".

- Torna-se apenas público o nome do tipo, deixando com que os seus constituíntes sejam privados ao módulo criador, e portanto, podendo ser manipulados, externamente, somente por meio de operações públicas do mesmo módulo.

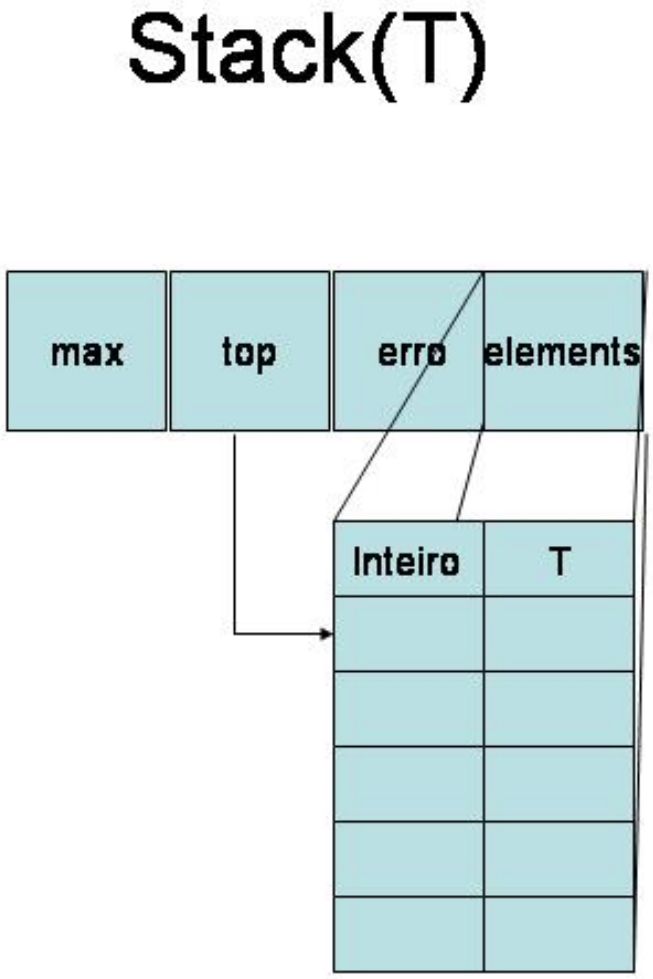
- Tendo declarado o tipo e as operações que atuam sobre o *TAD* e sendo esses públicos a outros módulos, este módulo contém um tipo abstrato de dados (*TAD*) de acordo com as definições, ou seja, um tipo que tem um conjunto único de operações que atua sobre ele, todos implementados em um único módulo.

Arquitetura de um *TAD* em *Machĭna*

```
module nome-do-modulo
      import elementos importados
      include elementos includos
      algebra:
            elementos declarados (funes e tipos)
      abstractions:
            declarao de abstraes (aes e iteraes)
end nome-do-modulo
```

TADs Implementados

- Lista Simplesmente Encadeada,
- Fila de Arranjo Circular,
- Pilha,
- Árvore Binária de Pesquisa,
- Árvore Patricia,
- SBB
- Tabela Hash.


Stack(T)
max
top
erro
elements
Inteiro
T

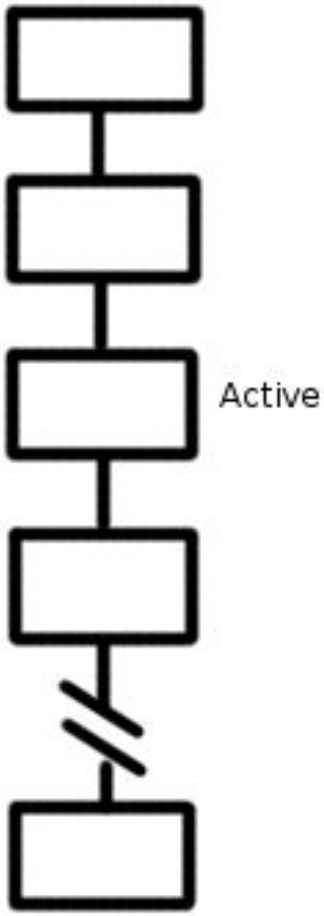
public type Stack(T) = **tuple** (elements: Int -> T, top: Int, erro: Bool, max: Int);

- `initStack`(s: Stack(T)): cria e inicializa uma pilha.
- `push`(s: Stack(T), in x: T): Insere o elemento que passado pelo parâmetro `x` no topo `top` da pilha.
- `pop` (s: Stack(T), out x: T): Retira o elemento do topo da pilha e devolvê-o em `x`.
- `empty` (in s: Stack(T), out vazia: Bool): Verifica se a pilha em questão está ou não vazia, devolvendo o resultado da verificao.
- `status` (in s: Stack(T), out houveErro: Bool): Retorna a informação sobre o estado gerado pela última operação sobre a pilha.

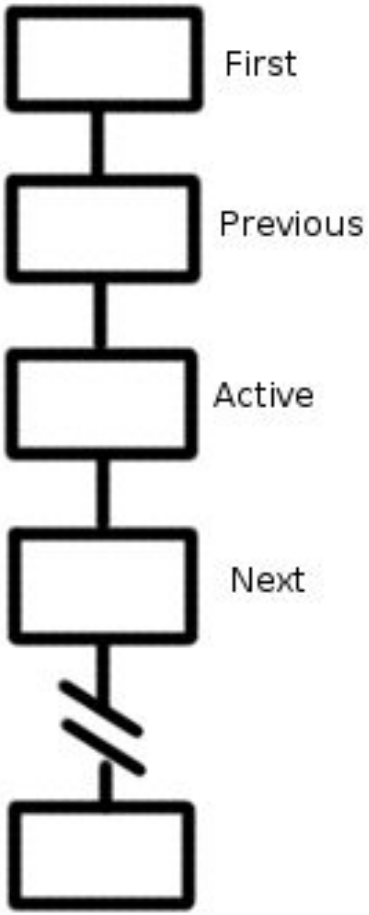
```
public action initStack(s: Stack(T), max: Int) is
     s.max:= max;
     s.top:= 0;
     s.erro:= \textbf{false};
end initStack

public action pop(s: Stack(T), out x: T) is
   loop:
       step 1: if (s.top $>$ 0) then
                   x:=  s.elements(s.top);
                   s.erro:= false;
               else s.erro:= true;
                   return;
               en
       step 2: s.top:= s.top -1;
end pop
```

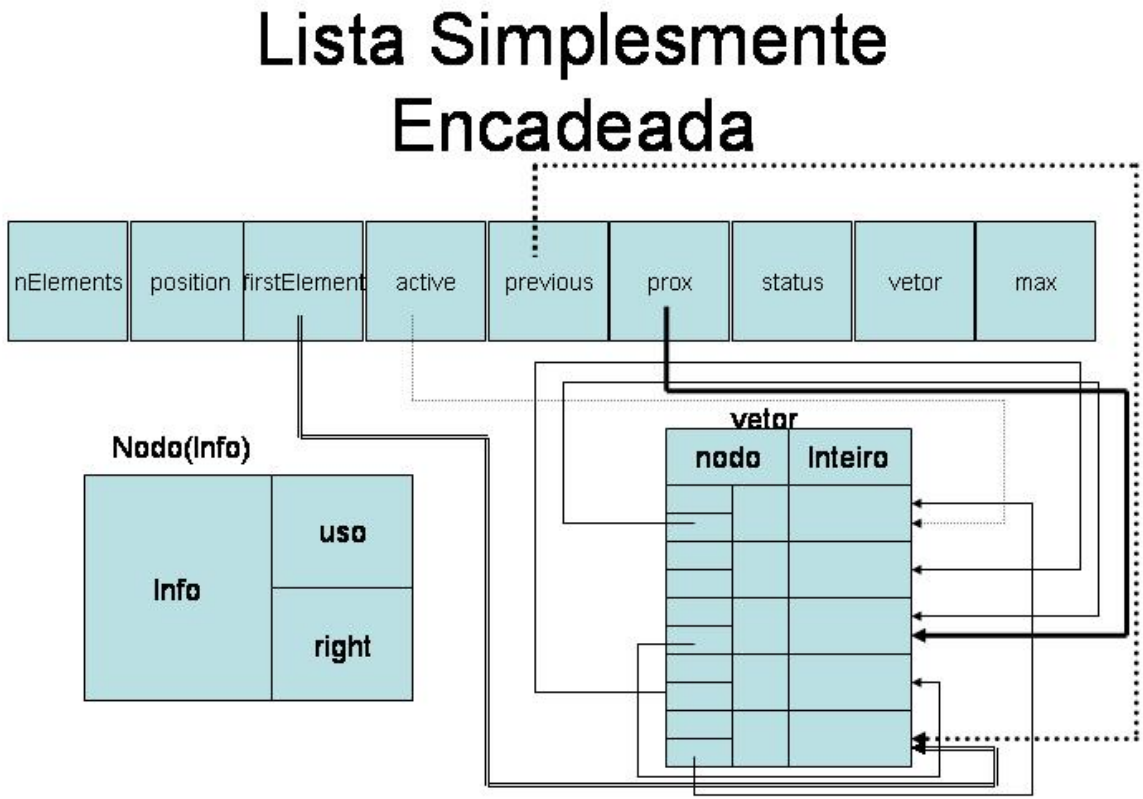
TAD Lista em *Machĭna*

Visão do Usuário



Visão do Implementador




Lista Simplesmente Encadeada
nElements
position
firstElement
active
previous
prox
status
vetor
max
Nodo(Info)
Info
uso
right
vetor
nodo
Inteiro

public type Lista(T) = **tuple**(nElements: Int, position: Int, firstElement: Int, active: Int, previous: Int, prox: Int, status: Erro, vetor: Element(T), max: Int);

type Nodo(T) = **tuple** (info:T, right: Int, uso: Bool);

public type Element(T) = Int -> Nodo(T);

public type Lista(T) = **tuple**(nElements: Int, position: Int, firstElement: Int, active: Int, previous: Int, prox: Int, status: Erro, vetor: Element(T), max: Int);

- `status`(in z: Lista(T), out erro: Bool): informa se houve algum erro durante a execução da última operação sobre a lista z.
- `checkRight`(in z:Lista(T), out offRight: Bool): informa se o elemento corrente está além do último elemento da lista z.
- `checkLeft`(in z:Lista(T), out offLeft: Bool): informa se o elemento corrente está aquém do primeiro elemento da lista z.
- `checkEmpty`(in z: Lista(T), out empty: Bool): verifica se a lista z se encontra vazia.

... *TAD* Lista em *Machĭna*

- `numOfElements`(in z: Lista(T), out nElement: Int): devolve a quantidade de elementos que a lista `z` possui.

- `getValue`(in z: Lista(T), out v: T): devolve o valor da informação de interesse do elemento corrente da lista `z` em `v`. Testa antes se a posição corrente se encontra dentro dos limites da lista, caso contrário gera um estado de erro.

- `changeValue`(z:Lista(T), in v: T): recebe uma lista `z` e um valor válido `v` para elementos da lista, substituindo o valor do elemento corrente pelo o valor dado.

- `checkFirst`(in z: Lista(T), out first: Bool): dada uma lista `z`, informa se o elemento corrente é o primeiro elemento da lista.

- `checkLast`(in z: Lista(T), out last: Bool): informa se o elemento corrente é o último elemento da lista `z`.
- `start`(z:Lista(T)): marca o primeiro elemento da lista `z` como sendo o elemento corrente.
- `forth`(z: Lista(T)): caminha uma posição para frente na lista `z`, caso seja possível, senão gera um estado de erro.
- go(z: Lista(T), in pos: Int): dada a posição desejada `pos`, marca como elemento corrente da lista `z`, o elemento correspondente a essa posição, caso seja possível, senão gera um estado de erro.

- `back`(z: Lista(T)): marca como corrente o elemento anterior ao atual corrente da lista `z`, se possível, senão gera um estado de erro.

- `search`(z: Lista(T), in v: T, out achou): dada uma informação válida `v`, caminha na lista `z` até encontrar o elemento contendo o valor `v`, devolvendo na função `achou` o resultado da pesquisa. Marca como elemento corrente, o elemento que foi encontrado ou o último elemento da lista, caso a informação passada como parâmetro no esteja na lista.

- `insertRight`(z: Lista(T), in v: T): recebe uma informação válida `v` e insere-a à direita do elemento corrente da lista `z`. Faz o elemento inserido o novo elemento corrente.

- `insertLeft`(z: Lista(T), in v: T): dada uma informação vlida `v` e insere-a à esquerda do elemento corrente da lista `z`.

- `finish`(z: Lista(T)): marca o elemento corrente como sendo o último elemento da lista z.
- `delete`(z: Lista(T)): retira o elemento corrente da lista z.
- `inicializaLista(z: Lista(T), in max: Int)`: cria uma lista z, iniciando seus valores.

Exemplo *TAD* Lista em *Machĭna*

```
public action forth (z: Lista(T)) is
        algebra
          vazia: Bool;
        loop:
          step 1: checkEmpty(z, vazia);
          step 2: if vazia then
                              z.status:= ErrorEmptyList;
                              return;
                  end
          step 3: if (z.position = z.nElements + 1) then
                     z.status:= ErrorOffRight;
                     return;
                  end
```

Exemplo *TAD* Lista em *Machĭna*

```
        step 4: if ( z.position = 0 ) then
                    start(z);
                    z.status:= NoError;
                    return;
                end
        step 5: if not ( z.position = 0 ) then
                    z.previous:= z.active;
                end
        step 6: if not ( z.position = 0 ) then
                     z.active:= z.prox;
                end
        step 7: if z.prox!= 0 then
                    z.prox:= vetor(z.prox).right;
                end
        step 8: z.position:= z.position + 1;
                z.status:= NoError;
end forth
```

Trabalhos Futuros

- Expansão da biblioteca de *TADs*;
- Expansão da biblioteca em outros aspectos;

Conclusão - Contribuições

- Estudo de *ASM*;
- Estudo da Linguagem *Machĭna*;
- Construção da biblioteca em *Machĭna*;

Perguntas

Apêndice - *ASM*

Apêndice - *Machĭna*