Tipos Abstratos de Dados em Ambiente Baseado em Máquina de Estados Abstratas

Projeto de Iniciação Científica

Prof. Roberto da Silva Bigonha

Belo Horizonte, 26 de novembro de 2004

1 Motivação

Um Tipo Abstrato de Dados (TAD) pode ser definido como um modelo matemático com um conjunto de operações definidas sobre o modelo. O conjunto de inteiros mais as operações de adição, subtração, etc, caracterizam um tipo abstrato de dados. Para implementar o modelo matemático utilizamos estruturas de dados [1, 12, 13].

Existem diversas vantagens em se utilizar Tipos Abstratos de Dados (TADs). A principal é o aumento do grau de modularidade do programa. Com TADs pode-se:

1. Permitir que se "esconda" como uma determinada estrutura é implementada.
2. Permitir que, no caso de se alterar a implementação de uma determinada estrutura de dados, as interfaces, ou seja, as declarações dos cabeçalhos dos procedimentos e funções não sejam alteradas.
3. Permitir uma maior reutilização de programas.
4. Permitir organizar programas maiores e mais complexos de uma forma ordenada e controlada.

Estas vantagens ficam mais evidentes à medida que mais programas, sejam eles pequenos ou grandes, simples ou complexos, são desenvolvidos e têm um reflexo direto na **qualidade** do produto final, ou seja, do seu programa. Portanto, a motivação básica para se implementar qualquer Tipo Abstrato de Dados (TAD) decorre naturalmente das vantagens descritas acima.

Os conceitos de TAD podem ser estendidos ao campo de especificação formal de programas ou de linguagens. Neste sentido, tipos abstratos de dados correspondem ao conceito de Sistemas Algébricos usados em modelos formais de definição de semântica de linguagens de programação. Estes sistemas algébricos permitem elevar o nível de abstração das especificações formais, encapsulando, via a filosofia do TAD, detalhes de sua definição.

A disponibilização de um conjunto de TAD ou implementações de sistemas algébricos fundamentais é de grande utilidade para especificação formal de sistemas de computação.

2 Objetivos

O objetivo deste projeto é prover uma biblioteca de TAD para implementar as estruturas de dados básicas [5, 6, 1, 12, 13], listas lineares, árvores e grafos como tipo abstrato de dados utilizáveis em especificação formal de sistemas.. A filosofia a ser adotada baseia-se no conceito de Máquina de Estados Abstratas(ASM) [16] e a ferramenta proposta para a implementação é AsmL [15] [14] . Como subproduto, este projeto dará ao bolsista a possibilidade de desenvolver estudos e trabalhos na área dos fundamentos teóricos da Computação, que irão contribuir para sua formação.

3 Máquinas de Estados Abstratas

Com o objetivo de prover um contexto para o detalhamento do projeto proposto, é apresentado a seguir uma pequena introdução ao modelo formal das Máquinas de Estados Abstratas.

ASM é um modelo formal de especificação de algoritmos, baseado no conceito de estado, descrito por uma álgebra, e de transição de estados. A descrição do estado inicial da máquina é então um sistema algébrico consistindo em vocabulário, universo de valores e interpretação inicial dos símbolos do vocabulário.

As mudanças de estado na máquina são dadas por uma regra de transição cuja função é modificar as interpretações de símbolos do vocabulário da álgebra, dando origem a um novo estado, que é então descrito por uma nova álgebra.

Uma ASM A é uma coleção finita de nomes de funções, cada uma com uma aridade fixa. Um *estado* de A é um conjunto não vazio, o superuniverso, junto com interpretações dos nomes da assinatura

de funções sobre os elementos do superuniverso. As interpretações são designadas *funções básica*s do estado. As funções básicas podem ser alteradas à medida que A muda de estado, mas o superuniverso se mantém inalterado.

Formalmente, supondo um superuniverso X, uma função básica de aridade r é uma função $X^r \rightarrow X$. Quando $r = 0$, a função é designada *elemento distinto*. O superuniverso sempre contém os elementos distintos `true`, `false` e `undef`, definidos como *constantes lógicas*. O elemento `undef` é utilizado para representar funções parciais, por exemplo, $f(\bar{a}) =$ `undef` significa que f é indefinida para a tupla $\bar{a}$. Uma relação r-ária sobre X pode ser vista como uma função $X^r \rightarrow \{$`true`, `false`$\}$. Um *universo* U é um tipo especial de função básica: uma relação unária geralmente identificada pelo conjunto dos elementos x tais que $U(x) =$`true`, i.e., $\{x : U(x)\}$.

Um programa de A é uma regra de transição, que pode, em sua forma mais simples, ser *instrução de atualização*, *construtor de bloco* ou *construtor condicional*.

Uma instrução de atualização tem o formato $f(\bar{t})$ `:=` t_0, onde f é o nome de uma função da assinatura de A, $\bar{t}$ é uma tupla de termos cujo tamanho é igual à aridade de f, e t_0 é outro termo. Os termos não possuem variáveis livres e são construídos recursivamente usando-se nomes de elementos distintos e aplicação do nome de uma função a outros termos. De maneira informal, a semântica da atualização acima é a seguinte: a tupla $\bar{t}$ é avaliada, e o valor da função básica f aplicada à tupla é alterado para o valor da avaliação de t_0. Ou seja, o nome f passa a ter uma nova interpretação.

Um construtor condicional tem genericamente a forma:

```
if g0 then R0
elseif g1 then R1
...
elseif gk then Rk
endif
```

A semântica do comando condicional é a seguinte: a regra R_i, $0 \leq i \leq k$, é executada se os termos booleanos $g_0, ..., g_{i-1}$ são avaliados para `false`, e g_i é avaliado para `true`.

Um construtor de bloco é um conjunto de regras. Sua semântica é a seguinte: todas as possíveis atualizações das regras contidas no bloco são disparadas em paralelo. Se uma atualização contradiz outra, uma escolha não-determinista é realizada.

Uma execução de um programa de A é uma seqüência de estados, onde o estado seguinte é obtido a partir do anterior através da execução da regra de transição do programa. A maioria das implementações de ASM determina o final da execução quando o disparo da regra de transição não produz nenhuma atualização. Esta forma de execução do modelo ASM difere substancialmente da natureza seqüêncial dos dos modelos reais de Computação Imperativa e enseja a possibilidade de se provar propriedades do programa. Entretanto, requer um estilo de programação inteiramente novo, justificando esta proposta de trabalho.

4 Abordagens Existentes para Implementação de TADs

4.1 Implementação usando Linguagens tipo Pascal

Existem na literatura algumas propostas de implementação de estruturas de dados básicas como tipo abstrato de dados. Uma delas [1] mostra uma implementação usando uma linguagem de programação procedural (Pascal), para atingir seu objetivo. Esta abordagem apresenta alguns problemas sérios. Tomemos como exemplo, o tipo abstrato de dados Lista. Para criar um **tipo abstrato de dados Lista**, é necessário definir um conjunto de operações sobre os objetos do tipo Lista. Por exemplo:
(1) Criar uma lista linear vazia.
(2) Inserir um novo item imediatamente após o i-ésimo item.
(3) Retirar o i-ésimo item.

(4) Localizar o i-ésimo item para examinar e/ou alterar o conteúdo de seus componentes.
(5) Combinar duas ou mais listas lineares em uma lista única.
(6) Partir uma lista linear em duas ou mais listas.
(7) Fazer uma cópia da lista linear.
(8) Ordenar os itens da lista em ordem ascendente ou descendente
(9) Pesquisar a ocorrência de um item com um valor particular em algum componente.

```
const
   InícioArranjo = 1;
   MaxTam      = 1000;
type
   Apontador = integer;
   TipoItem  = record
                 Chave : TipoChave;
                 {outros componentes}
               end;
   TipoLista = record
                 Item     : array [1..MaxTam] of TipoItem;
                 Primeiro : Apontador;
                 Último   : Apontador;
               end;
```

Se observarmos o modo como foi declarado o objeto `TipoItem` acima vemos que para o usuário utilizar este TAD Lista ele tem que ter pleno conhecimento dos detalhes da representação dos tipos declarados. Este fato não está associado com a linguagem usada, mas sim com a forma como o TAD foi definido. Neste exemplo, o usuário tem que saber que o registro `TipoItem` deve ter um campo `Chave` e que é permitido acrescenter *outros componentes* no mesmo. O item *outros componentes* é dependente do programa do usuário. Este fato contrasta com as principais vantagens sobre o uso de TADs citadas na Seção 1, pois permite que o indivíduo insira dentro da declaração de tipos, outros componentes. É fato que linguagens tipo Pascal não oferecem o recurso para implementar TADs, tudo fica em aberto. Embora a implementação seja um pouco complicada, ela poderia ser feita de tal forma que a informação dentro da declaração de tipos não fosse usada.

4.2 Implementação de TADs usando Linguagens Orientadas por Objetos

Outra abordagem [7] encontrada na literatura utiliza-se dos conceitos de programação orientada por objetos para implementar uma classe que engloba as estruturas de dados básicas como tipo abstrato de dados. Um exemplo desta abordagem utiliza uma linguagem orientada por objetos `Eiffel` [7] e define a classe de listas lineares como representações de seqüências, onde cada elemento de uma seqüência, denominado célula, é chamado de *Linkable*. Cada *Linkable* contém um valor e um apontador para outro elemento.

Neste modelo, T é o tipo dos valores (*value*). A lista é representada por uma célula separada chamada cabeça. A célula cabeça contém o endereço da primeira célula *linkable*. Esta célula pode também possuir outros itens, como por exemplo, um contador do número de elementos da lista (`nb-elements`).

A vantagem desta representação é que tanto a inserção quanto a remoção é rápida se houver um apontador para a célula *linkable* imediatamente à esquerda do ponto de inserção ou remoção. Por outro lado, esta representação não é muito boa para pesquisar por um elemento dado o seu valor ou posição, porque estas operações requerem um caminhamento seqüencial na lista. Existem duas estruturas de dados que são usadas normalmente para se implementar listas: arranjos e apontadores. Os arranjos são bons para acessar uma posição, mas ruins para inserções e remoções.

Para esta implementação são necessárias duas classes: uma para listas (*header*) e outra para os elementos da lista (*linkable*). A abordagem descrita utiliza as classes *LINKED-LIST* e *LINKABLE*. Note que a noção de *LINKABLE* é fundamental para a implementação, mas não é relevante para o usuário. Neste esquema é permitido ao usuário acesso ao módulo com primitivas para manipular listas

mas sem força-lo a se preocupar com detalhes de implementação, como por exemplo a presença de elementos *LINKABLES*.

Esta abordagem também apresenta alguns problemas. A classe *LINKED-LIST* mostra que estruturas que manipulam apontadores podem ser perigosas, principalmente quando combinados com ciclos. O uso de asserções auxilia um pouco, mas a dificuldade que aparece com este estilo de programação é um forte argumento para encapsular estas operações de uma vez por toda em módulos reusáveis, como é favorecido em abordagens orientadas por objetos.

Um outro aspecto preocupante da classe *LINKED-LIST* é a presença de redundâncias significativas em algumas operações. Para uma abordagem que enfatiza o reúso, este método como está implementado não é a melhor opção. Note que este problema é um problema de implementação, interno à classe, mesmo assim constitui um fato representativo em um problema de natureza mais séria, que é a interface da classe.

5 Trabalho Proposto

Nos dois métodos apresentados, o maior problema está no modo de tratar listas. Neles, uma lista é vista como um receptáculo passivo contendo informações. Para proporcionar ao usuário um produto mais adequado à suas aplicações é necessário tornar listas mais ativas, ou seja, fazer com que ela se "lembre" da última operação efetuada. A filosofia do trabalho proposto é olhar os objetos como máquinas com estados e introduzir procedimentos ou comandos que mudam o estado, e funções, ou *queries* nos estados. Esta abordagem produz uma interface que além de simples é mais eficiente que as outras mostradas neste texto.

Esta é a filosofia de implementação que será desenvolvida, no ambiente de programação AsmL, uma biblioteca de TAD contemplando as estruturas de dados básicas particularmente, listas, árvores e grafos.

5.1 Ferramenta Utilizada

AsmL [15, 14] é uma linguagem de especificação executável baseada o modelo ASM. Ela é útil em qualquer situação onde é preciso uma maneira precisa e não ambígua para especificar um sistema, tanto de software quanto de hardware. Uma especificação AsmL é uma forma ideal para comunicação de decisões de projeto para grupos de trabalho.

A linguagem AsmL possui recursos avançados, o que garante muito poder ao programador. Uma de suas características mais importantes está relacionada com o controle de visibilidade que permite tornar os elementos públicos, protegidos ou privados. Característica esta indispensável para a ocultação das estruturas de dados na implementação de TADs.

Outro recurso importante disponibilizado por AsmL é a existência de classes que possibilita uma organização modular do código. Em AsmL, objetos atuam como identidades que distinguem uma variável da outra. Objetos são elementos abstratos que não possuem estado ou outra característica qualquer. Apenas a transição da máquina provê a noção dinâmica de estado ao modelo.

6 Conclusão

Este trabalho terá como resultado, além das bibliotecas de TAD em AsmL, um manual do usuário, que será publicado na forma de monografia.

References

[1] Ziviani, N. *Projeto de Algoritmos com Implementação em Pascal e C*, Editora Pioneira, 1992.

[2] Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *Data Structure and Algorithms*, Addison-Wesley, 1983.

[3] Horowitz, Ellis and Sahni, Sartaj., *Fundamentals of Data Structures* Sixth Printing - Computer Science Press, Inc., 1976.

[4] Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, second edition - Prentice Hall-Software Series, 1988.

[5] Knuth, D., *The Art of Computer Programming, Volume 1: Fundamental Algoritms*, Addison-Wesley, Second Edition, 1973.

[6] Knuth, D., *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Second Edition, 1973.

[7] Meyer, Bertrand, *Object-oriented Software Construction*, Prentice-Hall International Series in Computer Science, C.A.R. Hoare Series Editor, 1988.

[8] TOOL – The Object Oriented Language – Programming for $Windows^{TM}$ MADE EASY, Comercializado por SPA 1995.

[9] Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley Publishing Company, Second Edition, 1991.

[10] Torgerson, Thomas W., *Visual Basic Professional 3.0 Programming*, Wiley-Oed Enterprise Computing, 1994.

[11] Yourdon, Edward, *Object-Oriented Systems Design – An Integrated Approach*, Yourdon Press Computing Systems, 1994.

[12] Wirth, N., *Algorithms and Data Structures*, Prentice-Hall, 1986.

[13] Wirth, N., *Algoritmos e Estruturas de Dados*, Prentice-Hall do Brasil LTDA,1989.

[14] Introducing AsmL: A Tutorial for the Abstract state Machine Language, Foundations of Software Engineering Microsoft Research Microsoft Corporation, may, 2002.

[15] AsmL: The Abstract State Machine Language, Foundations of Software Engineering Microsoft Research Microsoft Corporation, 2002.

[16] Gurevich, Y.; Evolving Algebras 1993: Lipari guide. In E. Bvrger, editor, Specification and Validation Methods, pages 9-63; Oxford University Press, 1995.

Implementação de Tipo Abstrato de Dados em Ambiente baseado em Máquina de Estados Abstratas

Projeto de Iniciação Científica

Prof. Roberto da Silva Bigonha

1. **Plano de Orientação de Iniciação Científica**

A duração deste trabalho está prevista por um ano, março/2005 a fevereiro/2006.

2. **Descrição das Etapas**

Etapa 1: Contextualização

1. Estudo de ASM e da ferramenta AsmL.
2. Estudo das estruturas de dados básicas.

Etapa 2: Desenvolvimento da Biblioteca de TAD

1. Definição da interface das classes.
2. Especificação do projeto e sua implementação.

Etapa 3: Testes e Depuração

1. Testes.
2. Depuração.

Etapa 4: Produção da documentação

1. Manual do usuário.
2. Manual do sistema.

Etapa 5: Relatório Final

1. Elaboração do Relatório Final.

3. **Cronograma**

Etapa 1: 15 de março a 15 de abril de 2005.

Etapa 2: 16 de abril a 15 de novembro de 2005.

Etapa 3: 16 de novembro a 20 de dezembro de 2005.

Etapa 4: 16 de abril de 2005 a 30 de janeiro de 2006.

Etapa 5: 01 a 28 de fevereiro de 2006.

Implementação de Tipo Abstrato de Dados em Ambiente baseado em Máquina de Estados Abstratas

Projeto de Iniciação Científica

Prof. Roberto da Silva Bigonha

1. Metodologia de Acompanhamento e de Avaliação

Na primeira etapa deste projeto o aluno se dedicará ao estudo das ferramenta AsmL e da metodologia ASM. Este estudo será dirigido pelo professor orientador em duas reuniões semanais com a duração de uma hora cada uma.

As demais etapas do projeto serão acompanhadas pelo professor orientador em uma reunião semanal com a duração de 1:30h aproximadamente durante todo o período de duração do projeto. O objetivo deste esquema é permitir ao bolsista a possibilidade de desenvolver os estudos e trabalhos de forma independente, contribuindo assim para sua formação. Esta metodologia será empregada durante todo o desenvolvimento do trabalho.