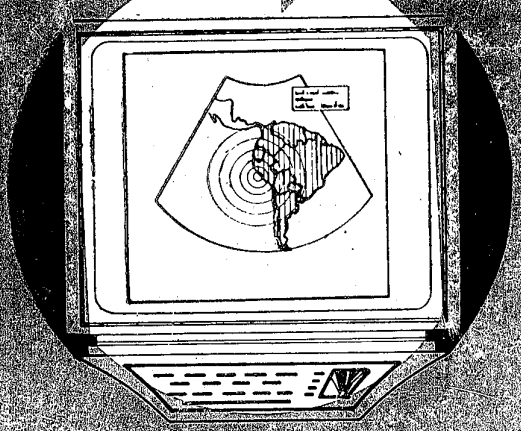
PANEL INFO '82

IX CONFERENCIA LATINOAMERICANA DE INFORMATICA

Centro Latinoamericano de Estudios de Informática (CLEI)
Asociación Peruana de Computación é Informática (APCI)
Universidad Nacional de Ingeniería (UNI)



16 AL 20 DE AGOSTO DE 1,982
LIMA - PERU

III Convención Peruana de Computación é Informática

PANEL INFO'82

Anales de la
IX Conferencia Latinoamericana
de Informática

Centro Latinoamericano de Estudios de Informática (CLEI)

y

III Convención Peruana de Informática

Organizado por:

Asociación Peruana de Computación é Informática (APCI)
Universidad Nacional de Ingeniería (UNI)

Patrocinado por:

Oficina Intergubernamental para la Informática (IBI)

Federación Internacional del Proceso de la Información (IFIP)

UNESCO

16 al 20 de Agosto de 1,982

LIMA - PERU

TEORIA DE TIPOS ABSTRATOS DE DADOS PARA PROGRAMACAO

UM ENFOQUE LOGICO

P.A.S. Veloso
Departamento de Informática
Pontificia Universidade Católica do RJ
Rio de Janeiro - Brasil

F.E.P. Pessoa
Núcleo de Processamento de Dados
Universidade Federal do Ceará
Fortaleza - Brasil

T.S.E. Maibaum
Departament of Computing
Imperial College of Science and Technology
Londres - Inglaterra

1. INTRODUÇÃO

A finalidade primordial deste trabalho é introduzir um novo enfoque para o conceito de tipo abstrato de dados e seu emprego em programação. Este enfoque é motivado pela prática de programação e parece ser mais adequado para ela.

Abstração é uma ferramenta bastante útil ao programador. Em particular, o uso de tipos abstratos de dados (TAD's) permite fatorar o programa em uma parte abstrata , que manipula objetos dos TAD's por meio de suas operações,e uma parte de implementação, que representa os objetos e operações abstratas por meio de outros mais concretos [LZ'74]. O emprego, talvez iterado, desse ponto de vista dá ao programa uma estrutura elegante e permite a fatoração natural de várias tarefas de programação: especificação, desenvolvimento, documentação, verificação, testes, etc.

Para realmente se tirar proveito da abstração,é necessário que os tipos de dados sejam especificados de maneira formal e independente de qualquer particular representação [LZ'75]. Várias maneiras de se apresentar uma especificação formal de um TAD têm sido propostas: axiomática, algébrica (inicial e terminal), etc. [H'72, GTW'78, GH'78].Entretanto, apesar de suas vantagens, cada um desses pontos de vista apresentam alguns inconvenientes.

O enfoque proposto neste trabalho tem por obje-

tivo exatamente aliviar tais inconvenientes. Mais especificamente:

- ser mais simples e próximo da prática de programação;
- ser adequado para o desenvolvimento e verificação de programas;
- permitir especificações incompletas;
- tratar de maneira mais flexível erros/exceções, parametrização e implementação.

As principais características do enfoque proposto são:

- nomeabilidade: cada objeto de um TAD deve ter um nome para poder ser referenciado por um programa;
- não-unicidade: alguns detalhes podem ser deixados em aberto;
- lógica: a linguagem do cálculo de predicados de 1ª ordem é mais flexível que a algébrica.

No que se segue, inicialmente, procura-se mostrar que não-unicidade e especificações incompletas são bastante naturais, bem como adequadas para a verificação de programas. Feito isso, alguns conceitos e aspectos de fundamentação teórica são apresentados, e, finalmente, o enfoque proposto é comparado com outros.

2. NATURALIDADE

Considere o TAD conjuntos (finitos) de elementos de D, cujas operações (com suas interpretações pretendidas intuitivamente descritas) são:

- ins [ins(s,d) insere d em s];
- rem [rem(s,d) remove d de s, caso d esteja em s];
- esc [esc(s) fornece um elemento de s, caso s não seja vazio].

Além disso, dispõe-se de uma constante phi para representar o conjunto vazio e dos seguintes predicados:

- vaz? [vaz?(s) testa se s é vazio];
- pert?[pert?(d,s) testa se d pertence a s].

Com este TAD podemos escrever o seguinte (trecho de) programa abstrato, onde x e y são variáveis declaradas do tipo Conj(D) e d é do tipo D:

```
while ¬ vaz?(y)
    do
        d:=esc(y);
        y:=rem(y,d);
        x:=ins(x,d)
    od;
```

Intuitivamente se percebe que, ao final da execução deste programa (que realmente termina), teremos y=phi e x=A∪B, onde A e B são, respectivamente, os valores iniciais de x e y.

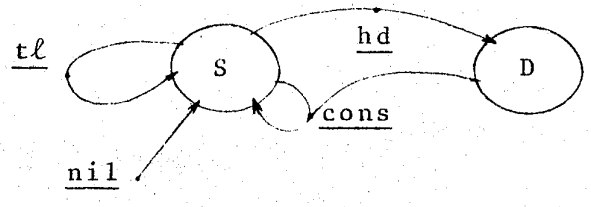
Note-se que não está claro quanto vale esc(phi), o que é totalmente irrelevante para o programa apresentado. Além disso, não é feita qualquer referência ao modo como esc seleciona o elemento devolvido. Porém, isso também é irrelevante: o que realmente importa é que esc(s) seja um elemento de s, caso s não seja vazio. Dito de outra forma, tudo que uma especificação de Conj (D) precisa informar é que ¬ vaz?(s) ⟹ pert?(esc(s),s).

Na verdade torna-se até difícil tentar explicar quem é esc(s) sem entrar em considerações sobre como s está representado, uma vez que esc é uma escolha não determinística (na terminologia lógica, esc é uma função de Skolem associada à fórmula ∀s∃d(¬ vaz?(s) ⟹ pert?(d,s)) [S'67]). Isso porém, é o que realmente importa a nível abstrato.

Uma conseqüência importante do não determinismo de esc é que, dado inicialmente x=A e y=B, podem-se garantir os valores finais x=A∪B e y=phi, porém, nada se pode dizer (nem isso é importante) acerca de seus valores intermediários. Dificuldades em se prever etapas intermediárias ocorrem também em programas com paralelismo (é fácil ver que as duas últimas atribuições do programa apresentado podem ser executadas em paralelo).

3. ADEQUAÇÃO

Considere, agora, o caso de seqüências(finitas) de elementos de um conjunto D, com as operações usuais cons, hd, tℓ e a constante nil. Abstraíndo-se dos detalhes de representação, tem-se o TAD Seq(D). Sua especificação formal consiste de duas partes, uma sintática e outra semântica. A primeira pode ser expressa, como usual, pelo diagrama abaixo:



A parte semântica, no enfoque proposto, consiste dos seguintes axiomas:

A1. $\neg\ \underline{cons}(d,s) = \underline{nil}$
A2. $\underline{cons}(d,s) = \underline{cons}(d',s') \Longrightarrow d=d' \wedge s=s'$
A3. $\underline{t\ell}(\underline{cons}(d,s)) = s$
A4. $\underline{hd}(\underline{cons}(d,s)) = d$,

onde d e d' são variáveis que percorrem o sorte D, s e s' percorrem o sorte S, ficando os quantificadores universais implícitos.

Os quatro axiomas acima expressam propriedades de seqüências segundo o entendimento intuitivo usual. Outra dessas propriedades é dada pelo seguinte axioma de indução (na verdade, este é uma conseqüência dos axiomas anteriores e da nomeabilidade):

A0. $\forall s(s=\underline{nil} \ \vee \ \exists s'\ \exists d' s=\underline{cons}(d',s'))$

Suponha, agora, que se deseje um programa para calcular o comprimento de seqüências. Admitindo-se os números naturais com suas operações costumeiras, quer-se um programa para calcular uma função $\ell g: S \longrightarrow \underline{Nat}$, especificada pelas seguintes propriedades:

L1. $\ell g(\underline{nil}) = 0$
L2. $\ell g(\underline{cons}(d,s)) = 1 + \ell g(s)$

O (trecho de) programa que se segue, onde as variáveis c e x são declaradas como do tipo Nat e S, respectivamente, intuitivamente cumpre este propósito.

```
c :=0 ;
while   ¬ x= nil
      do
            c:= c + 1;
            x:= tℓ(x)
      od;
```

Para verificar formalmente que este trecho de programa, referenciado por P, está de acordo com sua especificação, deve-se demonstrar que "se inicialmente x=A (um objeto arbitrário de S), então P termina (terminação), e então $x=\ell g(A)$ (correção parcial)".

Uma técnica usual para demonstrar a correção parcial [M'74] é associar à malha um invariante (No caso, $c+\ell g(x)=\ell g(A)$) e mostrar as seguintes condições de verificação:

V1. $x=A \wedge c=0 \Longrightarrow c + \ell g(x) = \ell g(A)$
V2. $\neg\ x= \underline{nil} \wedge c + \ell g(x) = \ell g(A) \Longrightarrow (c+1)+\ell g(\underline{t\ell}(x))=\ell g(A)$
V3. $x=\underline{nil} \wedge c+\ell g(x)=\ell g(A) \Longrightarrow c=\ell g(A)$

Para se fazer isso, usam-se as propriedades dos

naturais 0+n=n=n+0 e (m+1)+n = m+(1+n), além de se aplicar L2 à seguinte conseqüência de A0 e A3:

$$Cp.\ \neg s = \underline{nil} \implies \exists s' \exists d'\ (s = \underline{cons}(d', s') \wedge \underline{t\ell}(s) = s')$$

Com isso demonstra-se a correção parcial de P. Para se verificar a terminação de P, uma técnica usual é a do conjunto bem fundado [M'74]. Aqui, é suficiente verificar que

Te. Se $x \neq \underline{nil}$, então $\underline{t\ell}(x)$ é um subtermo de x,

o que decorre de Cp.

Alguns pontos devem ser ressaltados no exemplo em apreço. Primeiro, para se verificar a terminação não foi necessário se procurar um conjunto bem fundado:a relação de ser subtermo já se apresenta naturalmente. Segundo, na verificação parcial fez-se uso somente (além da especificação do programa e propriedades dos naturais) da especificação do TAD, pois esta é a única informação disponível sobre um tipo abstrato. Terceiro, procedendo-se de maneira análoga poder-se-iam verificar programas para inverter, concatenar,etc, seqüências.

Em resumo, a especificação dada é suficiente para muitos propósitos. Observe-se, porém, que ela é incompleta: a única informação dada sobre $\underline{t\ell}(\underline{nil})$ é (pela sintaxe) que este é de sorte S, deixando-se em aberto qual o seu valor.

4. FUNDAMENTAÇÃO

Considere um programa fatorado em uma parte abstrata e uma parte de implementação, como mencionado na introdução. Suponha que, usando-se a(s) especificação(ões) do (s) TAD(s), foi verificado que o programa abstrato está correto, como na seção precedente. O que se deve esperar - e exigir - da implementação? A resposta natural parece ser: garantir a correção do programa como um todo. Este critério simples e natural é o ponto chave do conceito de TAD aqui proposto.

O emprego de operações apenas parcialmente especificadas dá mais liberdade para a implementação, uma vez que um TAD pode, então, ser visualizado como uma estrutura com partes ou aspectos apenas delineados. Note-se,porém,que não se quer considerar as operações, ainda que parcialmente especificadas, como funções parciais. Isso dificultaria a intuição (que faz corresponder funções parciais a procedimentos que nem sempre param) e a fundamentação (haveria necessidade de uma lógica de funções parciais). Parece mais natural encarar uma estrutura com aspectos em aberto como uma classe de estruturas.

Tendo em vista estas motivações, são bastante razoáveis as definições que se seguem:"uma especificação é

um conjunto Σ de sentenças de uma linguagem de 1ª ordem; o TAD especificado por Σ é a classe de todos os modelos nomeados de Σ; um modelo nomeado é um em que cada objeto é denotado por um termo sem variáveis".

Apesar da ênfase dada a especificações incompletas, é claro que especificações completas também são permitidas. Com efeito, o que é realmente importante é o relacionamento entre programas e TAD's através de suas especificações. Para um tipo privado, criado unicamente para um programa particular, que, por exemplo, antes de tomar hd(k) testa se k é nil, não há por que se especificar um (único) valor para hd(nil). Já para um tipo público, que seria colocado numa biblioteca à disposição de vários usuários, seria conveniente se especificar hd(nil)=erro (Note-se que mesmo nesse último caso parece não ser útil se especificar a propagação de erros). Assim, para um tipo público a especificação tende a ser suficientemente completa [GH'78] ou mesmo completa [GTW'78], enquanto que para um tipo privado, a incompletude pode ser uma vantagem.

Uma das vantagens do enfoque proposto é que uma única ferramenta técnica de lógica, o conceito de extensão conservativa [S'67], parece ser a chave para abranger conceitos aparentemente díspares como parametrização, enriquecimento, implementação, etc. Em particular, implementação é basicamente uma interpretação em uma extensão conservativa. Assim, definir uma implementação do TAD Conj(D) especificado por Σ no TAD Seq(D) especificado por Δ consiste, basicamente, em dar uma interpretação de Σ em uma extensão conservativa de Δ (esta última não precisa definir esc completamente, apenas o suficiente para garantir a interpretação).

5. CONCLUSÃO

O enfoque proposto neste trabalho para TAD's é motivado pelo seu emprego em programação. Sua característica principal é considerar um TAD como uma classe de estruturas: os modelos nomeados da especificação.

Essa característica se contrapõe aos enfoques algébricos que encaram um TAD como único, a menos de isomorfismo [GTW'78]. Mesmo nos casos em que várias estruturas são admitidas, ainda se utilizam especificações suficientemente completas [GH'78], que não dão a flexibilidade de se escolher que detalhes deixar em aberto.

No que concerne a provas de correção de especificação e de implementação, os enfoques algébricos exigem que se prove uma propriedade de isomorfismo, o que costuma ser difícil. Na prática, contudo, geralmente uma propriedade de satisfação é suficiente. E é basicamente uma propriedade como esta, mais fácil de ser verificada, que o enfoque aqui proposto exige.

Uma outra característica deste enfoque, no seu objetivo de se aproximar da prática de programação, é sua

ênfase na sintaxe (especificações como teorias) ao invés da semântica (modelos). Um exemplo disso é a definição de implementação baseada no conceito sintático de interpretação de teorias em vez de transformações em estruturas.

Especificações incompletas já apareceram na literatura, se bem que com motivações ligeiramente distintas. A idéia de se completar uma especificação durante a fase de implementação é proposta por [PL'79]. Por outro lado no estudo de esquemas de programas, classes de interpretações têm sido consideradas em [CN'78].

Concluíndo, temos razão para crer que o enfoque aqui proposto apresenta vantagens sobre os outros usualmente encontrados na literatura. Ele parece permitir uma teoria de TAD's simples e bem fundamentada. As próximas etapas no desenvolvimento desta teoria seriam examinar mais detidamente os vários relacionamentos entre especificações sintáticas e suas contrapartidas semânticas intuitivamente desejáveis.

REFERÊNCIAS

[CN'78] B. Courcelle, M. Nivat - The Algebraic Semantics of Recursive Program Schemes; IRIA, Rapp.Rech. nº 300, 1978.

[GH'78] J.V. Guttag, J.J. Horning - The Algebraic Specification of Abstract Data Types; Acta Informatica, vol.10, p.27-52, 1978.

[GTW'78] J.A.Goguen, J.W. Thatcher, E.G.Wagner - An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types, em R.T. Yeh (ed.) Current Trends in Programming Methodology, vol. IV, Prentice-Hall, Englewood Cliffs,1978.

[H'72] C.A.R. Hoare - Proof of Correctness of Data Representations; Acta Informatica, vol. 1, p.271 - 281, 1972.

[LZ'74] B. Liskov, S. Zilles - Programming with Abstract Data Types; Proc. of a Symp. on very High Level Languages, SIGPLAN, vol 9, nº4, p.50-59, 1974

[LZ'75] B. Liskov, S.Zilles -Specification Techniques for Data Abstraction; IEEE Transactions on Software Engineering, vol.1, nº1, p.7-19, 1975

[M'74] Z. Manna - Mathematical Theory of Computation ; McGraw-Hill, New York, 1974.

[PL'79] T.H.C. Pequeno, C.J.P. Lucena - An Approach to Data Type Specification and Its Use in Program Verification; Infor. Proc. Letters, vol.8,nº2,p.98-103 1979.

[S'67] J.R. Shoenfield - Mathematical Logic; Addison-Wesley, Reading, 1967.

TEORIA DE TIPOS ABSTRATOS DE DADOS PARA PROGRAMAÇÃO: UM ENFOQUE LÓGICO

RESUMO

Este trabalho pretende introduzir um novo enfoque para o conceito de tipos abstratos de dados e seu uso em programação. Tal enfoque é motivado pela prática de programação e parece ser mais adequado para ela.

Tipos abstratos de dados têm sido apontados como uma ferramenta poderosa para a programação, permitindo a fatoração de um programa em uma parte abstrata e uma de implementação. Para se tirar real proveito dessa ferramenta é necessário, contudo, o uso de uma linguagem formal para a sua especificação. Para tal fim, têm sido usados enfoques algébricos, os quais, porém, apresentam alguns inconvenientes que o enfoque proposto objetiva aliviar.

É característica do presente enfoque o uso de modelos nomeados, não necessariamente um único, especificados em linguagem lógica, possibilitando especificações incompletas, o que é mais próximo da prática de programação.

O presente trabalho, após mostrar, através de exemplos, que o enfoque proposto é natural e adequado para o desenvolvimento de programas, indica como este enfoque pode ser fundamentado nas noções lógicas de extensão conservativas e de interpretação de teorias.