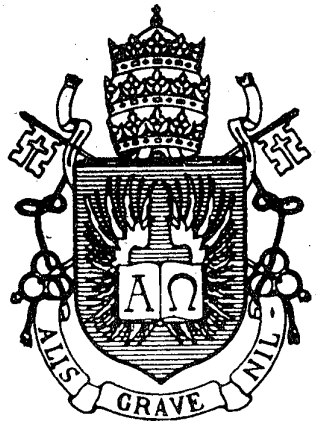


PUC

Série : Monografias em Ciência da Computação

Nº 20/83

INTRODUÇÃO À ESPECIFICAÇÃO E IMPLEMENTAÇÃO DE TIPOS ABSTRATOS DE DADOS

por

Francisco Edson P. Pessoa

e

Paulo Augusto S. Veloso

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 – CEP-22453

RIO DE JANEIRO – BRASIL

Série : Monografias em Ciência da Computação

Nº 20/83

Editor: Antonio L. Furtado Novembro de 1983

INTRODUÇÃO À ESPECIFICAÇÃO E IMPLEMENTAÇÃO DE TIPOS ABSTRATOS DE DADOS*

por

Francisco Edson P. Pessoa+

e

Paulo Augusto S. Veloso

* Trabalho realizado com auxílio financeiro da FINEP e do CNPq

+ Em licença do Núcleo de Processamento de Dados, Universidade Federal do Ceará, Fortaleza, CE.

RESUMO

Este trabalho se propõe a fornecer uma introdução didática aos conceitos básicos de especificação e implementação de tipos abstratos de dados encarados sob o enfoque algébrico. Tipos abstratos de dados são uma ferramenta útil e poderosa para o desenvolvimento de programas confiáveis. Seu emprego fatora o programa em duas partes: um programa, dito abstrato, que manipula objetos do tipo abstrato de dados unicamente por meio de suas operações, e um módulo de implementação, que representa os objetos e operações deste tipo em termos de outro(s), mais concreto(s). Uma especificação do tipo abstrato de dados deve descrever o comportamento de suas operações sem se prender a nenhuma representação específica. Especificação e implementação são tratadas aqui do ponto de vista algébrico, sem se prender a linguagens de programação.

O capítulo introdutório apresenta algumas considerações sobre o emprego de abstração no desenvolvimento de programas e o conceito de tipo em programação. O capítulo II trata de especificações de tipos abstratos de dados, primeiro sem erros, depois com erros, do problema da correção da especificação e apresenta uma metodologia para construir especificações corretas. O capítulo III trata de implementação: o conceito básico, o problema da correção e sugere uma metodologia para verificar a sua correção. O capítulo final tece algumas breves considerações sobre o relacionamento de especificações e implementações com a verificação de programas e sobre tipos abstratos de dados em linguagens de programação. Finalmente, o apêndice fornece alguns detalhes mais precisos sobre a especificação de tipos abstratos de dados como álgebras iniciais.

Palavras Chave : tipos abstratos de dados, especificação, implementação, correção, metodologia, enfoque álgebrico, equações, álgebras poli-sortidas, álgebra inicial.

ABSTRACT

This paper presents a didactical introduction to the basic concepts of specification and implementation of abstract data types within an algebraic approach.
Abstract data types are a powerful tool for the development of reliable programs. By employing abstract data types the program is factorized into two parts, namely a program, called abstract, manipulating objects of the abstract data type solely by means of its operations, and an implementation module, which represents the objects and operations of this data type in terms of other, more concrete, data type(s). A specification of an abstract data type must describe the behavior of its operations without resorting to any specific representation.
Specification and implementation are examined here from an algebraic viewpoint without consideration of programming language aspects.

The introductory chapter presents some remarks on the usage of abstraction in program development and on the concept of type in programming. Chapter II deals with specification of abstract data types, first the error-free case, then including errors, the problem of specification correctness, and presents a methodology for the construction of correct specifications. Chapter III deals with implementation: the basic concept, the problem of correctness and suggests a methodology for verifying the correctness of implementations.

The final chapter presents some brief remarks on the relationship between program verification and specifications and implementations as well as on abstract data types in programming languages. Finally, the appendix makes more precise some details concerning the specification of abstract data types as initial algebras.

Key words: abstract data types, specification, implementation, correctness, methodology, algebraic approach, equations, many-sorted algebras, initial algebra.

SUMÁRIO

CAPÍTULO I

INTRODUÇÃO

O conceito de tipo é de fundamental importância em programação. Qualquer programador sabe que em um programa, cada variável, constante ou expressão tem associado a ela um único tipo. Em PASCAL, por exemplo, a associação de um tipo a uma variável é feita pela sua declaração de tipo; em FORTRAN, tal associação é deduzida da primeira letra do nome da variável, caso esta não tenha sido explicitamente declarada.

Um tipo de dados é basicamente uma coleção de valores munida de um conjunto de operações e testes. O tipo Boolean, em PASCAL, por exemplo, tem como valores o conjunto {false, true} e como operadores $\vee$, $\wedge$ e $\neg$ para indicar a disjunção "ou", conjunção "e" e negação, respectivamente. Embora tipos diferentes possam ter o mesmo símbolo de operador (exemplo: + para os tipos inteiro e real) numa linguagem, tais símbolos são interpretados como operações diferentes, podendo essa diferença ser detectada em tempo de compilação.

Associado a um tipo também está uma representação para os valores do tipo, uma vez que, sem uma representação, não teríamos como descrever o comportamento de suas operações. A representação do tipo inteiro numa linguagem de programação está implícita, de modo que o programador não se preocupa

com ela , mas as operações a nível de máquina foram definidas em termos dessa representação. Assim sendo, a especificação da representação também determina o conjunto de valores de um tipo , uma vez que ela é definida em termos de tipos existentes (cada inteiro é representado por uma sequência finita de bits, e seus operadores definidos com base nessa representação).

O conceito de tipo não é uma novidade surgida com a teoria das linguagens de programação. Na realidade os matemáticos e lógicos estão familiarizados com esse conceito. No raciocínio matemático é comum fazer-se distinção entre funções reais , funções complexas, conjunto de funções etc. De fato, quando um matemático introduz no seu raciocínio uma nova variável, ele costuma ter o cuidado de explicitar seu tipo.

Exemplos:

"Seja f uma função de duas variáveis reais"

"Seja S uma família de conjuntos de inteiros"

Abstração se refere ao processo mental pelo qual, quando em confronto com um conjunto de objetos, situações ou processos, podemos reconhecer suas similaridades, concentrando-nos sobre estas e ignorando suas diferenças. Para citar um exemplo , consideremos o conjunto {7, 1.33, 'A', 0.5 E6} que contém valores de três tipos providos por quase toda linguagem de programação : inteiro, real e caráter. Olhando o conjunto mais abstratamente podemos dizer que ele contém apenas dois tipos: número e caracter . De um ponto de vista mais abstrato ainda, o conjunto tem apenas um tipo: constante.

Comum a todas as ciências, em ciência da computação, contudo, o termo abstração tem adquirido, gradualmente, um significado mais específico, ao se referir ao desejo de se querer considerar um conceito independentemente de suas várias possíveis representações.

Um tipo abstrato de dados deve ser independente de sua representação, no sentido de que os detalhes de como ele é implementado são propositadamente "escondidos" de seus usuários . O usuário de um tipo abstrato é provido com certas operações para o tipo, necessitando saber apenas o que tais operações fazem e não como elas o fazem. O comportamento do usuário deve ser semelhante àquele que ele tem para com o uso de uma sub-rotina, que descreve uma função específica por meio de um algorítmo que lhe é desconhecido: no ponto em que ele invoca a sub-rotina, o detalhe relevante é o que a sub-rotina faz, sendo o como totalmente irrelevante. Da mesma forma, ao nível de implementação " é desnecessário complicar o como por considerações dos porquês, isto é, as razões para a invocação da sub-rotina não precisam ser consideradas por seu implementador" (Guttag [12]).

De um certo modo, os tipos providos por uma linguagem de programação (ex. inteiro, real, etc.) são abstratos, já que o programador faz uso deles sem se preocupar com sua representação. Porém o termo tipo abstrato de dados tem sido reservado para aqueles tipos que não são supridos pela linguagem, mas criados pelo usuário, fazendo uso dos mecanismos que a linguagem lhe oferece para esse fim. Tais tipos, normalmente, são usados num nível e realizados noutro mais baixo. Mas isso não se dá au

tomaticamente. Ao invés disso, um tipo abstrato de dados é realizado ao se escrever um programa que o define em termos da suas operações.

As linguagens de programação convencionais oferecem uma poderosa ferramenta para construção de abstrações que são as funções ou procedimentos. Linguagens como ALGOL 68 e PASCAL oferecem mecanismos que possibilitam a definição e uso de novos tipos. Em muitos casos, o novo tipo é definido em termos de outros previamente definidos, chamados de tipos constituintes ; os valores do novo tipo são estruturas de dados que tem como componentes os valores dos tipos constituintes, os quais podem ser selecionados e extraídos da estrutura. Melhores mecanismos para a definição de tipos abstratos de dados foram introduzidos com SIMULA 67 (Dahl-Hoare [02]) e sucessivamente aperfeiçoados em linguagens como CLU (Liskov-Zilles [23]), ALPHARD (Wulf-London - Shaw [40]) e ADA(Ledgard [21]).

As características salientes do conceito de tipo que mais interessam à comunidade de computação são assim sumarizadas (Hoare [18]):

i) Um tipo determina a classe de valores que podem ser assumidos por uma variável ou expressão.

ii) Todo valor pertence a um único tipo.

iii) O tipo de um valor denotado por uma constante, variável ou expressão pode ser deduzido de sua forma ou contexto, sem qualquer conhecimento prévio de

seu valor.

iv) Cada operador espera receber operandos de algum tipo fixo, e devolve um resultado de tipo fixo (usualmente o mesmo). Quando um mesmo símbolo é aplicado a diferentes tipos, esse símbolo pode ser visto como ambíguo, denotando diferentes operadores. A resolução de uma tal ambigüidade sempre pode ser conseguida em tempo de compilação.

v) As propriedades dos valores de um tipo e as operações primitivas definidas sobre eles são especificadas formalmente e de modo independente de representação por meio de axiomas.

vi) A informação de tipo numa linguagem de programação de alto nível é usada para prevenir ou detectar construções desprovidas de sentido num programa e para determinar o método de representação e manipulação de dados no computador.

Abstração é um processo inerente as aplicações dos computadores ao mundo real. A primeira atitude do programador quando do projeto de qualquer programa é concentrar-se nas características relevantes do problema, ignorando os fatores considerados irrelevantes. O uso de linguagens tradicionais exige que o passo seguinte do projeto seja a decisão de como representar no computador a informação abstrata. O passo final e então programar o computador para fazê-lo manipular as representações de dados de forma que essas manipulações levem ao efeito desejado no mundo real.

Confiabilidade e facilidade de entendimento são duas qualidades desejáveis de um programa . A programação estruturada é uma tentativa de disciplinar o processo de construção de programas de modo a se obter programas com essas características. De acordo com essa disciplina, um problema é resolvido por meio de um processo de sucessivas decomposições (Wirth [39]). Num primeiro passo o programador escreve um programa que resolve o problema, mas que opera sobre objetos abstratos, no sentido de que os mesmos não estão definidos na linguagem que será usada na codificação do programa (a conveniência disso está no fato de que o programador escolhe os objetos e operações que mais se adeqüem a solução do problema).

A tarefa seguinte do programador é procurar se convencer de que seu programa resolve corretamente o problema. Nesse ponto ele deve se preocupar apenas de como seu programa faz uso das abstrações, e não com qualquer detalhe de como elas serão realizadas. Quando satisfeito com a correção de seu programa o programador volta sua atenção para as abstrações criadas. Cada abstração representa um novo problema, requerendo solução. Essa solução pode ser dada em termos de novas abstrações. O problema estará completamente resolvido quando todas as abstrações forem realizadas na linguagem escolhida.

Como se depreende facilmente, o método de desenvolvimento de programas por refinamentos sucessivos, descrito acima, estimula o programador a deixar de lado a decisão de como representar os dados, para dedicar-se a elaboração de seu algorítmo, expresso como um programa abstrato, operando sobre dados abstratos. Isto feito, o programador escolhe para os dados uma repre

sentação concreta em termos de tipos diretamente ou quase diretamente representáveis na memória do computador, programando as operações primitivas requeridas pelo programa abstrato. Essa forma de fatorar a construção de programas num programa que manipula um tipo abstrato de dados e numa implementação do tipo abstrato em termos de uma representação selecionada, alivia, além de tornar construtiva a tarefa de verificação de correção do programa, também fatorada na verificação de correção do programa abstrato e na verificação de correção da representação de dados.

Como se pode notar, a programação com tipos abstratos de dados segue naturalmente a idéia de desenvolvimento de programas por refinamentos sucessivos, subvertendo a atitude tradicional de construção de programas descrita (abstração → representação → codificação).

Um aspecto importante de um tipo abstrato de dados se refere a sua especificação. Para se especificar um tipo é fundamental uma boa notação. Uma linguagem informal, tal como a linguagem natural, nem sempre é eficiente para criação ou comunicação de abstrações . De fato, somente com muito cuidado e esperteza é possível se escrever uma especificação precisa em linguagem natural. O problema é mais grave ainda quando a abstração tem que ser comunicada a alguém. A especificação pode não somente ser indefinida e portanto ambígua, como a própria linguagem na qual a especificação é feita, conter ambigüidades . Se a ambigüidade é percebida, esta pode ser resolvida. Porém o que normalmente ocorre é cada um dos envolvidos com a abstração formar sua própria concepção da abstração, criando assim um

sério problema de interface.

Naturalmente o uso de uma linguagem formal não nos garante que uma especificação não contenha ambigüidades ou seja consistente (é possível escrever gramáticas ambíguas com BNF). O que uma linguagem formal nos proporciona são critérios para o reconhecimento de ambigüidades e inconsistências, aumentando dessa forma a probabilidade do reconhecimento de falhas na especificação. A verificação de que a especificação de um tipo esta correta tem uma importância fundamental, uma vez que é contra esta que se vai testar a correção da implementação do tipo.

As técnicas para investigar a correção de programas podem ser classificados em formais e informais. As informais (depuração, teste etc), embora mais comumente usadas, investigam as propriedades do programa de forma incompleta, além de serem extremamente dependentes da habilidade e intuição do programador. Técnicas formais, como aquelas devidos a Floyd e Hoare (Manna [29]), por outro lado, tentam estabelecer propriedades do programa com respeito a todas as entradas válidas, por meio de um raciocínio em que cada passo é formalmente justificado por regras de inferência, axiomas e teoremas. Em tais técnicas, a especificação formal dos pressupostos e axiomas sobre os quais o programa se baseia desempenha papel fundamental na prova de correção, além de auxiliar o desenvolvimento de programas que sejam corretos por construção e de se constituir numa importante parte da documentação do programa.

Em Liskov e Zilles [26] foram descritos cinco métodos

para especificação de tipos abstratos de dados, tendo Pereda [33] agrupado-os em quatro categorias. Neste trabalho usaremos álgebras para especificar tipos abstratos de dados o que tem sido enfatizado como adequado em vários trabalhos (Zilles [42], Goguen-Thatcher-Wagner [10], Guttag-Horning [15]).

No que se segue enfocaremos os aspectos concernentes a especificação e implementação de tipos abstratos de dados. O capítulo II refere-se à especificação. Aí são abordados as metodologias para especificação e sua verificação, sendo os resultados teóricos, demonstração de lemas a teoremas, transferidos para um apêndice. Os problemas da implementação são abordados no capítulo III. Finalmente, o capítulo IV apresenta algumas breves considerações sobre os mecanismos de abstração providos por linguagens de programação e sobre o processo de programação com tipos abstratos de dados. Estes tópicos são examinados com mais detalhes e exemplos em [34].

CAPÍTULO II

ESPECIFICAÇÃO DE TIPOS ABSTRATOS DE DADOS

II.1 Introdução

Um tipo de dados, como já enfatizamos, é basicamente uma coleção de valores munida de um conjunto de operações definidas para o tipo. Se o tipo é para ser compreendido a um nível abstrato, isto é, independente de representação, a especificação do comportamento de suas operações não deve conter referências a qualquer possível representação.

A linguagem usada para se especificar um tipo abstrato é de fundamental importância. Tipos abstratos podem ter o comportamento de suas operações apresentado de maneira informal. Contudo o uso de uma linguagem informal, tal como a linguagem natural, raramente se mostra eficiente para criação e comunicação de abstrações. Como já foi enfatizado na introdução, é bastante difícil escrever uma especificação precisa em linguagem natural. O problema é mais grave ainda quando a abstração tem que ser comunicada a alguém (lembre-se de que o especificador de um tipo abstrato nem sempre é o seu implementador ou usuário), quando ambigüidades podem criar sérios problemas de interface.

Naturalmente o uso de uma linguagem formal não nos ga -

rante uma especificação livre de ambiguidades ou inconsistências. O que esta nos propicia são critérios para o reconhecimento destas falhas, aumentando assim as possibilidades de sua detecção na especificação. A garantia de que a especificação é correta é primordial uma vez que é contra esta que se tem que verificar a implementação do tipo abstrato.

Vários métodos têm sido propostos para especificação de um tipo abstrato de dados (Liskov e Zilles [26]). Dentre estes o método algébrico tem recebido grande atenção, sendo apresentado em muitos trabalhos (Zilles [42], Goguen-Thatcher-Wagner [10] e Guttag-Horning [15]) como adequado para especificação de um tipo abstrato. Segundo essa abordagem uma especificação para um tipo T consiste de uma descrição sintática e uma descrição semântica; a especificação sintática define os nomes, domínios e com domínios das operações de T, enquanto a especificação semântica contém um conjunto de axiomas, na forma de equações, relacionando as operações do tipo T, umas com as outras.

Guttag [15] resume assim as principais vantagens de uma especificação algébrica:

i) Ela é declarativa, evitando assim detalhes de programação e dependência de linguagem.

ii) Ela é uma descrição razoavelmente intuitiva do comportamento de várias estruturas.

iii) Ela é suficientemente rigorosa para permitir uma prova de que uma particular realização da estrutura de dados está de acordo com a especificação.

iv) Ela é fácil de se ler e compeender, facilitando assim uma verificação informal do fato de que está de acordo com as intenções de seu criador.

A seguir vamos apresentar a especificação algébrica abordando em detalhes o sistema de especificação e a correção da especificação. Completaremos o capítulo apresentando uma metodologia para construção de especificações.

II.2. Sistema de Especificação

Uma álgebra homogênea A é um par $<D,\Sigma>$, onde D, chamado do domínio da álgebra, é um conjunto não-vazio de valores, e Σ é um conjunto (finito) de operações finitárias $F: D^n \rightarrow D$. Exemplo: $<N, \{+, .\}>$ é uma álgebra que tem como domínio os naturais e como operações $F: N^2 \rightarrow N$ a soma e o produto.

Birkhoff e Lipson generalizam esse conceito para uma álgebra heterogênea. Uma álgebra heterogênea ou poli-sortida A, é essencialmente uma família de domínios A_s com uma coleção de operações (funções) entre eles. Tais álgebras surgem muito naturalmente em ciência da computação. O conjunto S de índices para os domínios da álgebra é chamado de conjunto de sortes ou nomes de tipos. Como exemplo de uma tal álgebra, seja:

Sortes: $S = \{Nat, Bool\}$

Símbolos de Operações: $\Sigma = \{Z,F,V,SUC,PRED,MNOR\}$,

$Z : \rightarrow \underline{Nat}$

$F : \rightarrow \underline{Bool}$

$V : \rightarrow \underline{Bool}$

$SUC: \underline{Nat} \rightarrow \underline{Nat}$

$PRED: \underline{Nat} \rightarrow \underline{Nat}$

$MNOR: \underline{Nat} \times \underline{Nat} \rightarrow \underline{Bool}$

Obs: Se $F: \rightarrow S$, F é dita ser uma <u>constante</u> de sorte S.

É evidente que o exemplo acima não exibe uma álgebra específica, uma vez que falta, por exemplo, atribuir significado a cada uma das operações. Dito de outra forma, o que temos é meramente uma linguagem. Uma álgebra fica determinada quando se aponta um modelo para essa linguagem, ou seja, quando se diz especificamente o que são os domínios e qual o significado de cada operação. Como exemplo, apontamos o seguinte modelo que define a álgebra A dos naturais com as operações de <u>sucessor</u>, <u>predecessor</u> e <u>menor que</u>:

$A_{\underline{Nat}}$ = conjunto dos naturais ({0,1,2,3,..... })

$A_{\underline{Bool}}$ = conjunto de valores lógicos ({Falso,Verdadeiro})

Z_A = constante <u>0</u> (zero)

F_A = constante <u>Falso</u>

V_A = constante <u>Verdadeiro</u>

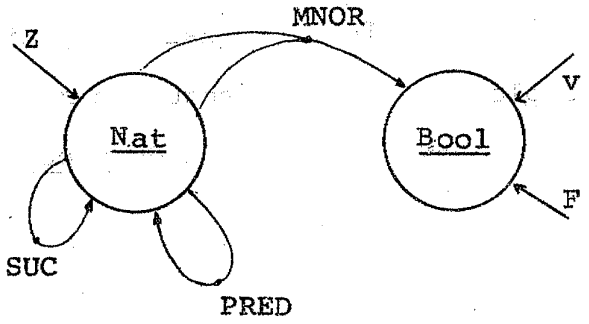
SUC_A = função sucessor $(n \in N \rightarrow SUC_A(n) = n + 1)$

$PRED_A$ = função predecessor $(n \in N \rightarrow \begin{cases} PRED_A(n) = n, \text{se } n=0 \\ PRED_A(n) = n-1, \text{se } n \neq 0 \end{cases})$

$MNOR_A$ = função lógica < (menor do que)

Do exposto fica claro que, para se especificar uma álgebra, é necessário uma parte sintática, a linguagem, e uma parte semântica, um modelo que interpreta os símbolos da linguagem. Fica claro também que para uma dada linguagem é possível haver várias álgebras, bastando para tal mudar o significado das operações, por exemplo, ou dessas e dos domínios.

Uma notação conveniente para se indicar a sintaxe de uma álgebra S-sortida é sugerida em Goguen-Thatcher-Wagner [10]. Segundo essa notação os domínios da álgebra são indicados por círculos e as operações indicadas por setas, com a origem indicando os tipos dos argumentos da operação e o destino o tipo do seu resultado. Para o exemplo em questão, teríamos:



De uma maneira mais formal e precisa vamos definir o que seja uma álgebra S-sortida, para o conjunto de sortes S dado.

Definição 1. Um domínio de operadores Σ para S é uma família $\Sigma_{w,s}$ de conjuntos, com $s \in S$ e $w \in S^*$ (S^* é o conjunto de todas as cadeias de símbolos obtidos de S,

inclusive a cadeia vazia λ). Se $F \in \Sigma_{w,s}$, dizemos que F é um símbolo operacional de aridade w e de tipo s.

Definição 2. Seja Σ um domínio de operadores. Uma Σ-álgebra A consiste de um conjunto $A_s \neq \phi$ para cada $s \in S$ (chamado o domínio de A para o tipo s) e uma função

$$\sigma_A: A_{s_1} \times A_{s_2} \times \ldots \times A_{s_n} \longrightarrow A_s$$

para cada símbolo $\sigma \in \Sigma_{w,s}$, com $w = s_1 s_2 s_3 \ldots s_n$ (chamando a operação de A denotada por σ). Para $\sigma \in \Sigma_{\lambda,s}$, $\sigma_A \in A_s$ (também denotado por $\sigma_A: \rightarrow A_s$) isto é, $\Sigma_{\lambda,s}$ é o conjunto de nomes das constantes de A de sorte s.

Álgebra dos Termos

Conforme enfatizamos, uma linguagem não fala de uma álgebra, mas de uma classe de álgebras. Para nos referirmos a uma álgebra específica necessitamos atribuir um domínio a cada $s \in S$ e interpretar nesses domínios cada símbolo de função. Para a linguagem introduzida no início exibimos uma álgebra A, a álgebra dos naturais com as operações de sucessor, predecessor e menor que. Vamos agora construir uma outra álgebra para mesma linguagem, a álgebra T dos termos. Tal álgebra é gerada de modo puramente sintático, com seus domínios sendo compostos das expressões ou fórmulas obtidas pela aplicação recursiva das operações aos termos gerados a partir das constantes (Universo de Herbrand). A álgebra T tem a seguinte forma:

Z_T = Z

F_T = F

V_T = V

SUC_T : Se $t \in T_{\underline{Nat}}$ então $SUC_T(t) = SUC(t)$

$PRED_T$: Se $t \in T_{\underline{Nat}}$ então $PRED_T(t) = PRED(t)$

$MNOR_T$: Se $t_1, t_2 \in T_{\underline{Nat}}$ então $MNOR_T(t_1,t_2) = MNOR(t_1,t_2)$

$T_{\underline{Nat}}$ = conjunto de expressões ou termos gerados a partir das constantes pela aplicação das operações que dão resultado no sorte Nat, ou seja:

{Z, SUC(Z), PRED(Z), SUC(SUC(Z)),
PRED(SUC(Z)), SUC(PRED(Z)), PRED(PRED(Z)),
.....}

$T_{\underline{Bool}}$ = conjunto de expressões ou termos obtidos a partir das constantes pela aplicação das operações que dão resultados no sorte Bool, ou seja

{F, V, MNOR(Z,Z), MNOR(Z,SUC(Z)),.....}

De maneira mais formal, a álgebra dos termos é a seguinte:

Domínios: i) $\Sigma_{\lambda,s} \subseteq T_s$

ii) Se $\sigma \in \Sigma_{w,s}$, $w = s_1 s_2 \ldots s_n$ e $t_i \in T_{s_i}$ então $\sigma(t_1,t_2,\ldots t_n) \in T_s$

iii) Os únicos elementos de T_s são os obtidos por (i) e (ii).

Operações: i) Para $\sigma \in \Sigma_{\lambda,s}$, $\sigma_T = \sigma \in T_s$

ii) Para $\sigma \in \Sigma_{w,s}$, $w = s_1, s_2, \ldots, s_n$ e $t_i \in T_{s_i}$, $\sigma_T(t_1,t_2,\ldots,t_n) = \sigma(t_1,t_2,\ldots,t_n) \in T_s$

Como se observa os elementos dos domínios T_s são construidos de modo puramente sintático ao se colocar o símbolo de função $\sigma \in \Sigma_{w,s}$ na frente dos termos que são os argumentos da função (os parenteses são usados somente para auxiliar o entendimento). A obtenção dos elementos de T_s se dá a partir das constantes $\sigma \in \Sigma_{\lambda,s}$ que são inicialmente colocados em T_s. As operaçoẽs tem definições apropriadas para gerarem os domínios.

Do modo como foi definido os elementos de um domínio T_s da álgebra T são todos diferentes, embora quando interpretados segundo um modelo alguns elementos possam representar um mesmo objeto. Como exemplo, vamos interpretar cada símbolo da linguagem de T segundo a álgebra A já apresentada. Assim sendo vários termos em T seriam identificados, como indicado a seguir:

	em A	
Z	⟶	$0 = Z_A$
SUC (Z)	⟶	$1 = SUC_A\ (Z_A)$
PRED (Z)	⟶	$0 = PRED_A\ (Z_A)$
SUC (SUC(Z))	⟶	$2 = SUC_A\ (SUC(Z_A))$
PRED (SUC(Z))	⟶	0
SUC (PRED(Z))	⟶	1
PRED (PRED(Z))	⟶	0
SUC (SUC(SUC(Z)))	⟶	3
PRED (SUC(SUC(Z)))	⟶	1
SUC (PRED(SUC(Z)))	⟶	1

$F \xrightarrow{\text{em A}} \text{Falso} = F_A$

$V \longrightarrow \text{Verdadeiro} = V_A$

$\text{MNOR}(Z,Z) \longrightarrow \text{Falso} = F_A$

$\text{MNOR}(Z, \text{SUCC}(Z)) \longrightarrow \text{Verdadeiro} = V_A$

Sejam t_1 e t_2 dois termos de T que quando interpretados são idênticos, ou seja, a equação $t_1 = t_2$ vale quando interpretada em A (Ex. $\text{SUC}(Z) = \text{SUC}(\text{PRED}(\text{SUC}(Z)))$). Cada equação $t_1 = t_2$ descreve o comportamento de diferentes sequências de operações. A lista de todas as equações válidas em T nada mais é que uma caracterização do comportamento das operações segundo o modelo da álgebra A. O inconveniente dessa descrição para as operações de A é que o conjunto de equações é infinito. É frequentemente possível, porém, dar-se uma descrição finita a esse conjunto através do uso de variáveis, como feito a seguir:

i) $\text{PRED}(Z) = Z$

ii) $\forall\, t \in T_{Nat},\ \text{PRED}(\text{SUC}(t)) = t$

iii) $\text{MNOR}(Z,Z) = F$

iv) $\forall\, t \in T_{Nat},\ \text{MNOR}(\text{SUC}(t), Z) = F$

v) $\forall\, t \in T_{Nat},\ \text{MNOR}(Z, \text{SUC}(t)) = V$

vi) $\forall\, t, t' \in T_{Nat},\ \text{MNOR}(\text{SUC}(t), \text{SUC}(t')) = \text{MNOR}(t,t')$

O conjunto de equações acima se constitui num conjunto de axiomas que permite identificar os termos de T sem recorrer à

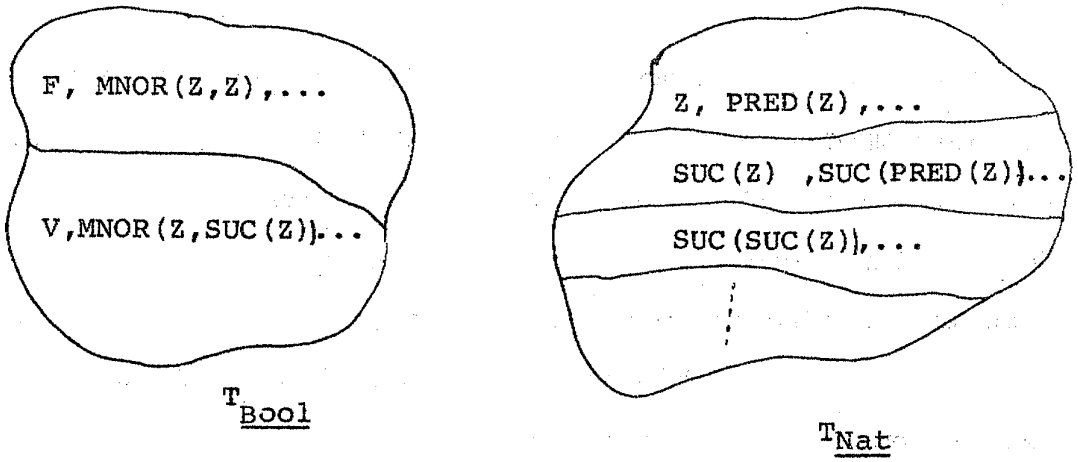
sua interpretação(Exemplo: PRED(SUC(Z)) = Z pela equação (ii) , portanto MNOR(Z, PRED(SUC(Z))) = MNOR(Z,Z) = F pela equação(iii)). Por outro lado, o fato de essas equações terem sido obtidas usando-se a imposição de que dois termos em T são idênticos quando suas interpretações em A também o são, faz com que esse conjunto de equações seja também satisfeito pela álgebra A. O conjunto de axiomas em questão se constitui numa especificação para o comportamento das operações do modelo A.

Álgebra Quociente

A igualdade de termos na álgebra T, segundo uma interpretação A, define em cada domínio T_s uma relação binária que é reflexiva (t = t), simétrica (se $t_1 = t_2$ então $t_2 = t_1$) e transitiva (se $t_1 = t_2$ e $t_2 = t_3$, então $t_1 = t_3$). Uma relação binária que é reflexiva, simétrica e transitiva é chamada de relação de equivalência.

Uma propriedade importante de uma relação de equivalência é que se R é uma tal relação sobre um conjunto C então nós podemos dividir C em subconjuntos disjuntos, chamados classes de equivalência, tal que x e y se relacionam segundo R se e somente se x e y pertencem a um mesmo subconjunto.

O conjunto de axiomas antes apresentado divide T_{Bool} em duas classes de equivalência e T_{Nat} em infinitas classes:



$T_{\underline{Bool}}$ $T_{\underline{Nat}}$

Observe que todos os termos de uma mesma classe de equivalência são iguais quando interpretados em A (os axiomas foram escritos com esse objetivo). Isto significa que, como argumento das operações, a substituição de um termo por outro que lhe é equivalente não muda o resultado da operação quando esta é interpretada em A. Uma relação de equivalência com essa propriedade de substituição é dita ser uma relação de congruência.

Para cada um dos domínios $T_{\underline{Bool}}$ e $T_{\underline{Nat}}$, vamos escolher um representante de cada classe de equivalência. Se o termo t é esse representante, denotaremos a classe que contém t por [t] .Com isso nós podemos construir uma nova álgebra, a álgebra quociente, denotada por $T/\equiv$, para a linguagem que temos utilizado para exemplo:

$T_{\underline{Nat}}/\equiv$ = conjunto de classes de $T_{\underline{Nat}}$, ou seja:

$$\{ [Z], [SUC(Z)], [SUC(SUC(Z))],...\}$$

$T_{\underline{Bool}}/\equiv$ = conjunto de classes de $T_{\underline{Bool}}$, ou seja:

$$\{ [F], [V] \}$$

$Z_{T/\equiv} = [Z] = \{Z, PRED(Z), PRED(SUC(Z)), PRED(PRED(Z)...\}$

$F_{T/\equiv} = [F] = \{F, MNOR(Z,Z), MNOR(SUC(Z), PRED(Z)),...\}$

$V_{T/\equiv} = [V] = \{V, MNOR(Z,SUC(Z)), MNOR(PRED(Z),SUC(Z)),..\}$

$SUC_{T/\equiv}$: Se $[t] \in T_{\underline{Nat}}/\equiv$ então $SUC_{T/\equiv}([t]) = [SUC_T(t)] =$

$= [SUC(t)]$

$PRED_{T/\equiv}$: Se $[t] \in T_{\underline{Nat}}/\equiv$, então $PRED_{T/\equiv}([t]) = [PRED_T$

$(t)] = [PRED(t)]$

$MNOR_{T/\equiv}$: Se $[t], [t'] \in T_{\underline{Nat}}/\equiv$ então $MNOR_{T/\equiv}([t],[t']) =$

$= [MNOR_T(t,t')] = [MNOR(t,t')]$

De um modo geral a álgebra $T/\equiv$ tem para domínios os conjuntos das classes de equivalência dos domínios da álgebra T, e cada operação tem como argumentos classes de equivalência e como resultado a classe de equivalência que contém o resultado da aplicação da correspondente operação da álgebra T aos representantes das classes que são argumentos, ou seja:

i) Se $\sigma \in \Sigma_{\lambda,s}$. então $\sigma_{T/\equiv} = [\sigma_T]$

ii) Se $\sigma \in \Sigma_{w,s}$, $w = s_1, s_2, \ldots s_n$, e $[t_i] \in T_{s_i}/\equiv$ então

$\sigma_{T/\equiv}([t_1], [t_2], \ldots, [t_n]) = [\sigma_T(t_1, t_2, \ldots, t_n)]$

Morfismos entre Álgebras

Dado duas Σ-álgebras A e B, por uma função $h: A \to B$ queremos indicar uma família de funções $\langle h_s: A_s \to B_s \rangle$ para cada $s \in S$.

Dentre as funções que se pode estabelecer entre as álgebras de uma mesma linguagem nós estamos particularmente interessados naquelas que preservam suas operações, os chamados morfismos.

Definição 3. Se A e B são duas Σ-álgebras, um Σ-homomorfismo $h:A \to B$ é uma família de funções $\langle h_s : A_s \to B_s \rangle$, $s \in S$, tal que as operações são preservadas, ou seja:

i) Se $\sigma \in \Sigma_{\lambda,s}$, então $h_s[\sigma_A] = \sigma_B$

ii) Se $\sigma \in \Sigma_{s_1,s_2,\ldots,s_n,s}$ e $\langle a_1,a_2,\ldots,a_n \rangle \in$

$A_{s_1} \times A_{s_2} \times \ldots \times A_{s_n}$, então $h_s[(\sigma_A(a_1,a_2,\ldots,a_n)] =$

$= \sigma_B (h_{s_1}[a_1], h_{s_2}[a_2],\ldots,[h_{s_n}] a_n)$

A composição de homomorfismos é também um homomorfismo. Para cada álgebra A, a identidade i_A é um homomorfismo. Um homomorfismo $h:A \to B$ é um isomorfismo se e somente se cada $h_s: A_s \to B_s$ é uma bijeção (um-a-um e sobre).

No início dessa seção apresentamos como exemplo uma linguagem e apontamos três álgebras para essa linguagem, A, T e $T/\equiv$. Que relação existe entre elas? Pode-se mostrar que existe um homomorfismo de T em A, e que $T/\equiv$ e A são isomorfas. Esse último resultado, por seu particular interesse será demonstrado na seção II.4.

É uma prática comum em álgebra abstrata identificar objetos isomorfos, isto é, tratá-los como idênticos. Desse modo as álgebras $T/\equiv$ e A são idênticas (a menos dos nomes dos elementos),

e ao falarmos de $T/\equiv$ estamos falando de A ou qualquer outra álgebra isomorfa aquela.

A conclusão importante que se tira desse resultado é que dada uma Σ-álgebra A cuja descrição (significado das operações e domínios) geralmente é feita de modo informal, pode-se contrapor uma outra Σ-álgebra, a álgebra quociente $T/\equiv$,que lhe é isomorfa, tendo porém sua descrição feita de maneira mais formal e completamente independente de representação. O conjunto de axiomas que dá origem a congruência $\equiv$ é satisfeito tanto pela álgebra A como pela álgebra $T/\equiv$ e se constitui desse modo numa <u>especificação</u> para o comportamento das operações da álgebra A.

<u>O tipo Abstrato e sua Especificação</u>

Para o exemplo que estamos explorando, os sortes envolvidos são <u>Nat</u> e <u>Bool</u>. O tipo que se pretende especificar no caso é os naturais com as operações SUC,PRED e MNOR. A descrição do tipo foi apresentada de maneira informal pela álgebra A e de maneira formal e <u>independente de representação</u> pelo conjunto de equações geradoras da congruência $\equiv$ necessária para tornar as álgebras A e $T/\equiv$ isomorfas.

Como já frisamos, álgebras isomorfas são idênticas.Desse modo, qualquer uma dentre todas as álgebras isomorfas a A serviria para descrever o tipo apresentadpor A. A escolha de $T/\equiv$ é devido ao fato de estadescrevê-lo de maneira formal e independente de representação.

A especificação é um conjunto ξ de equações que faz $T/\equiv$ isomorfa ao modelo que se quer especificar.

Definição 4. Uma especificação é uma tripla $<S,\Sigma,\xi>$, onde S é um conjunto de sortes, Σ é um conjunto de símbolos operacionais e ξ é um conjunto de Σ-equações.

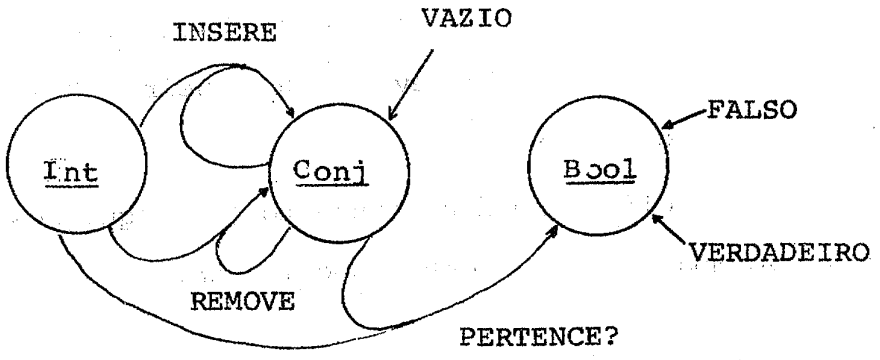
Para citar um exemplo, considere a especificação dos naturais com as operações sucessor, predecessor e menor que (a álgebra A que apresentamos anteriormente). Conforme já vimos, temos:

ξ = {i) PRED(Z) = Z
ii) PRED(SUC(x)) = x
iii) MNOR(Z,Z) = F
iv) MNOR(SUC(x),Z) = F
v) MNOR(Z,SUC(x)) = V
vi) MNOR(SUC(x), SUC(y)) = MNOR(x,y),

para x,y: naturais}

Um Exemplo Adicional

Um exemplo bem conhecido de um tipo abstrato é Conjuntos-de-Inteiros. Supondo os tipos referentes aos sortes pré-definidos, as operações a serem especificadas são: inserir ou remover um inteiro num conjunto e testar se um inteiro pertence a um conjunto. Gráficamente:



Um conjunto de axiomas que especifica esse tipo é o seguinte:

Para c: Conj e i,j:Int,

i) PERTENCE?(VAZIO,i) = FALSO

ii) PERTENCE?(INSERE(c,i),j) = Se i=j então VERDADEIRO
senão PERTENCE?(c,j)

iii) RETIRA(VAZIO,i) = VAZIO

iv) RETIRA(INSERE(c,i),j) = Se i=j então RETIRA(c,j)
senão INSERE(RETIRA(c,j),i)

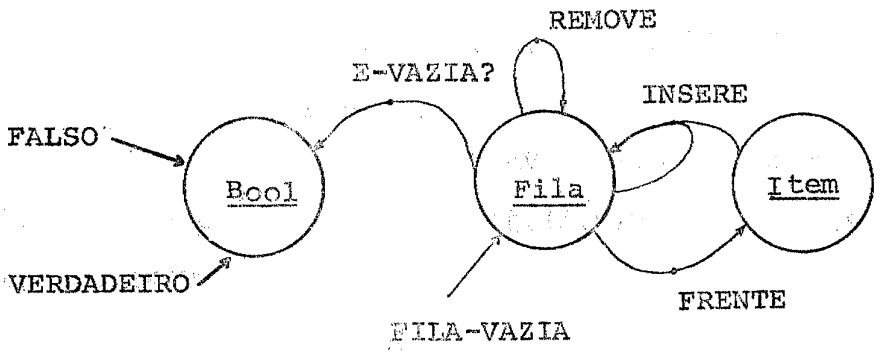
v) INSERE(INSERE(c,i),j) = Se i=j então INSERE(c,i)
senão INSERE(INSERE(c,j),i)

Nesse exemplo as equações (iii), (iv) e (v) são condicionais. Como mostra Goguen-Thatcher-Wagner [10] é possível escrever equações condicionais sem que se fuja ao escopo algébrico, pela introdução das equações com if-then-else. Essas equações assumem um papel importante na especificação algébrica,uma vez que nos permitem escrever definições recursivas para as operações.

II.3. Especificação com Erro

Os exemplos apresentados na seção anterior não continham erros. Isto, porém, não é o caso para a maioria dos problemas usuais, onde o tipo tem inerente a sua estrutura certos casos em que a aplicação de determinadas operações é sem sentido . Como exemplo seja acesar o topo de uma pilha vazia, dividir um número por zero, etc.

Para se ter uma idéia dos problemas que ocorrem ao se tratar algebricamente os casos que envolvem erros, vamos considerar o tipo Fila-de-Item, onde Item e Bool são pré-definidos:



Os axiomas que especificam esse tipo são (Gaudel [09]):

Para q: Fila e i:Item

i) REMOVE(INSERE(q,i)) = Se E-VAZIA?(q) então FILA-VAZIA
senão INSERE(REMOVE(q),i)

ii) FRENTE(INSERE(q,i)) = Se E-VAZIA?(q) então i
senão FRENTE(q)

iii) E-VAZIA?(FILA-VAZIA) = VERDADEIRO

iv) E-VAZIA?(INSERE(q,i)) = FALSO

Tais axiomas, contudo, não especificam completamente o tipo Fila-de-Item, uma vez que as operações FRENTE e REMOVE são indefinidas quando tem como argumento FILA-VAZIA. O modo natural de se resolver esse problema seria acrescentar os axiomas,

v) FRENTE(FILA-VAZIA) = $INDEF_i$

vi) REMOVE(FILA-VAZIA) = $INDEF_q$

onde $INDEF_i$ e $INDEF_q$ seriam elementos especiais (mensagens de erro) pertencentes aos sortes Item e Fila , respectivamente.

Infelizmente a introdução de tais axiomas dá origem a termos indesejáveis, tais como

q = INSERE(FILA-VAZIA,FRENTE(FILA VAZIA),

visto que FRENTE(q) = $INDEF_i$ e E-VAZIA?(q) = FALSO.

A introdução dos valores especiais $INDEF_i$ e $INDEF_q$, por outro lado, exige que se acresça novos axiomas afim de se estabelecer a propagação de erros, ou seja, qualquer operação que tenha como argumento um elemento $INDEF_t$ (pertencente ao sorte t) deve dar como resultado o valor INDEF do tipo apropriado. Assim os seguintes axiomas deveriam ser acrescido:

vii) INSERE($INDEF_q$,i) = $INDEF_q$

viii) INSERE(q,$INDEF_i$) = $INDEF_q$

ix) E-VAZIA?($INDEF_q$) = $INDEF_b$

x) FRENTE($INDEF_q$) = $INDEF_i$

xi) REMOVE($INDEF_q$) = $INDEF_q$

O grave, porém, é que a introdução desses novos axiomas vem acompanhada de inconsistências, como explicita o resultado a seguir: De acordo com o axioma (i) deduz-se que , pelo (iii)

REMOVE(INSERE(FILA-VAZIA,FRENTE(FILA-VAZIA) = FILA-VAZIA.

Por outro lado,pela aplicação dos axiomas (v),(viii) e (xi),temos

REMOVE(INSERE(FILA-VAZIA,FRENTE(FILA-VAZIA)) =

REMOVE(INSERE(FILA-VAZIA,$INDEF_i$)) =

REMOVE($INDEF_q$) = $INDEF_q$

Portanto,

FILA-VAZIA = $INDEF_q$

Porém isso implica que

E-VAZIA?($INDEF_q$) = VERDADE,

o que é uma contradição, pois por (vii) e (iv) temos

$$INDEF_q = INSERE(INDEF_q, i)$$

$$E\text{-}VAZIA?(INDEF_q) = FALSO$$

Na realidade, o sorte Fila seria colapsado para um único elemento, qual seja $INDEF_q$.

Uma forma de se evitar esse problema é se redefinir todos os axiomas com um teste para se saber se cada operando é ou não indefinido. Isto, contudo, resolve apenas o problema do sorte Fila. Reespecificar os tipos de sortes Item e Bool requer não só dificuldades adicionais, como é inaceitável do ponto de vista da filosofia básica de tipos abstratos de dados.O desejável no caso é manter o conjunto inicial de axiomas ((i),(ii),(iii) e (iv)) imutável,especificando a parte os casos de erros.Adicionalmente, os tipos pré-definidos usados na especificação devem ser protegidos, isto é, deve ser possível enriquecê-los com novos termos sem no entanto alterar a relação entre os termos já existentes.

A solução desse problema, de uma forma que mais se aproxima do que frequentemente ocorre em programação, é a sugestão de se acrescer à especificação uma parte de restrição onde os erros seriam especificados (Guttag-Horning [15]). Basicamente,existem dois tipos de restrições: Pré-condições e Falhas.

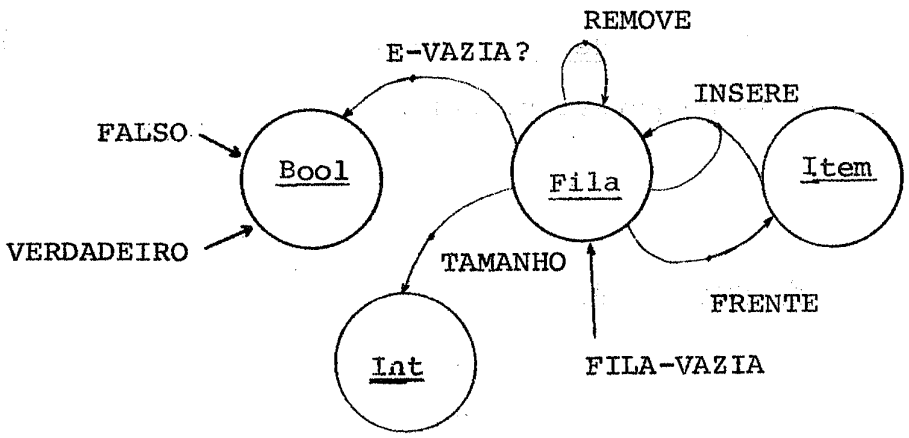
Uma pré-condição estabelece um limite na aplicabilidade dos axiomas e desse modo nos termos que o usuário do tipo pode escrever com a garantia de que os axiomas são satisfeitos. É tarefa do usuário verificar ou provar a pré-condiçãc e em provas sobre o tipo de dados ela é considerada uma hipótese. Ao usuário cabe também decidir e especificar o que deve ser feito quando a pré-condição não ocorre.

Uma falha, por outro lado, estabelece que qualquer representação para o tipo é suposta falhar nos casos expressos. Ca

be ao implementador do tipo verificar a validade dos termos e decidir o que deve ser feito quando uma falha ocorrer (parar, editar uma mensagem de erro, etc.).

A fim de ilustrar o que antes foi dito, vamos enriquecer o exemplo em questão adicionando uma restrição ao tamanho dos objetos do sorte Fila. Para tanto, nós necessitamos de uma nova operação, TAMANHO, e uma restrição que especifique que uma falha deve ocorrer se o tamanho da de uma fila é maior que 100. A aplicação das operações REMOVE e FRENTE, por seu turno, está sujeita a pré-condição de a fila não ser vazia.

A especificação completa do tipo é (Gaudel [09]):



Axiomas:

Para q: Fila e i: Item

i) REMOVE(INSERE(q,i) = Se E-VAZIA?(q) então FILA-VAZIA
senão INSERE(REMOVE(q),i)

ii) FRENTE(INSERE(q,i) = Se E-VAZIA?(q) então i
senão FRENTE(q)

iii) E-VAZIA?(FILA-VAZIA) = VERDADEIRO

iv) E-VAZIA?(INSERE(q,i) = FALSO

v) TAMANHO(FILA-VAZIA) = 0

vi) TAMANHO(INSERE(q,i)) = TAMANHO(q) + 1

Restrições:

i) Pre(REMOVE,q) = ¬ E-VAZIA?(q)

ii) Pre(FRENTE,q) = ¬ E-VAZIA?(q)

iii) TAMANHO(q) ≥ 100 → Falha(INSERE,q,i)

Goguen trata o problema de erros de forma diversa do apresentado aqui. Ao invés de remover os termos errados do tipo abstrato, ele os introduz no conjunto total dos termos. Dessa forma ele desenvolve uma álgebra de erros de modo muito similar à álgebra quociente da seção anterior([11]).

II.4. Correção da Especificação

A necessidade básica de se ter uma especificação para um determinado tipo de dado advém do fato de se querer utilizar a especificação na verificação da implementação do tipo. Dessa forma, é de fundamental importância o fato de se estar seguro quanto a correção da própria especificação.

Quando já existe um modelo matemático para o tipo de dado, uma prova rigorosa da correção da especificação algébrica é possível e necessária (este não é o caso para a maioria dos tipos abstratos definidos pelo usuário especialmente para serem usados em seu programa). Quando o modelo matemático não é intei

ramente bem definido, o processo de validação da especificação tem a virtude de poder contribuir para clarificá-lo melhor.Quando, no entanto, nenhum modelo matemático existe para o tipo, a validação da especificação cresce em dificuldade e importância.

Nessa seção vamos tratar apenas da correção de especificações algébricas de tipos para os quais existe um modelo matemático já definido. A idéia básica pode assim ser resumida: uma especificação $<S,\Sigma,\xi>$ é correta se a álgebra quociente $T/\equiv$ é isomorfa ao modelo matemático (lembramos que ξ é o conjunto de equações (axiomas) gerador da congruência $\equiv$).

Álgebra de Termos Canônicos

O método desenvolvido por Goguen (Goguen-Thatcher-Wagner [10]) para demonstrar a correção de especificações algébricas se baseia em representantes canônicos de cada classe de equivalência da álgebra $T/\equiv$. A idéia básica é se construir uma nova Σ-ξ-álgebra C (uma Σ-álgebra que satisfaz o conjunto de equações ξ), chamada de álgebra dos termos canônicos, cujos domínios C_s são conjuntos de termos canônicos, e então mostrar que C é isomorfa ao modelo matemático A que define o tipo que se quer especificar. Uma vez que C é isomorfa à $T/\equiv$ (o teorema que garante esse resultado, bem como aquele que garante a existência da álgebra C,está enunciado e demonstrado no apêndice), conclui-se que o modelo A é também isomorfo a $T/\equiv$ e, dessa forma, ξ é uma especificação correta de A.

Em princípio, a escolha dos termos canônicos é totalmen

te arbitrária, mas na prática é necessário se fazer sua seleção de um modo disciplinado a fim de se tirar proveito da estrutura dos termos na prova da correção. A definição que se segue provê essa disciplina.

> Definição 5 - Uma Σ-álgebra C é uma álgebra de Σ-termos canônicos se $C_s \subseteq T_s$ para cada $s \in S$, e se $\sigma\ (t_1,\ldots,t_n) \in C_s$, então $t_i \in C_{s_i}$ (sendo $\sigma: s_1,\ldots,s_n \to s$) e o $\sigma_C\ (t_1,\ldots,t_n) = \sigma\ (t_1,\ldots,t_n)$.

Um Exemplo

No início da seção II.2 apresentamos uma linguagem e explicitamos três modelos para ela: A álgebra A dos naturais com as operações de sucessor, predecessor e menor que, a álgebra T dos termos da linguagem, e a álgebra $T/\equiv$ das classes de equivalência obtidas de T pela congruência $\equiv$ gerada pelo conjunto de equações ξ lá expresso. No final daquela seção fizemos referência ao fato de $T/\equiv$ ser isomorfa a A. A demonstração desse fato segue basicamente a metodologia aqui apresentada, conforme passamos a ilustrar.

Seguindo o método apresentado, o que temos a fazer é construir, inicialmente, uma álgebra C de termos canônicos. Para tal, temos que escolher um representante para cada classe de equivalência e definir as operações da nova álgebra, tudo de acordo com o preconizado na definição 5. Assim, temos:

$C_{Nat} = \{SUC^{o}(Z), SUC^{1}(Z), SUC^{2}(Z), \ldots\ldots, SUC^{n}(Z), \ldots\}$,

onde $SUC^{o}(Z)$ é Z

$C_{Bool} = \{F, V\}$

$Z_C = Z$

$F_C = F$

$V_C = V$

$SUC_C(SUC^n(Z)) = SUC^{n+1}(Z)$, para $n \geq 0$

$PRED_C(Z) = Z$, e para $n \geq 1$ $PRED_C(SUC^n(Z)) = SUC^{n-1}(Z)$

$$MNOR_C(SUC^m(Z), SUC^n(Z)) = \begin{cases} V, & \text{se } m < n \\ F, & \text{se } m \geq n \end{cases}$$

O que temos que fazer agora é simplesmente mostrar que a álgebra C é isomorfa a álgebra A. De fato:

Seja $h: C \to A$ assim definida:

$h_{\underline{Bool}}(F) = \underline{Falso}$

$h_{\underline{Bool}}(V) = \underline{Verdadeiro}$

$h_{\underline{Nat}}(SUC^n(Z)) = n$

a) h é um <u>homomorfismo</u> de C em A.

<u>Prova</u>:

Seja $t \in C_{\underline{Nat}}$. Assim t é da forma $t = SUC^n(Z)$, para $n \geq 0$.

$$h_{\underline{Nat}}\left[SUC_C(t)\right] = h_{\underline{Nat}}\left[SUC^{n+1}(Z)\right] \quad \text{, pela def. de } SUC_C$$

$$= n+1 \quad \text{, pela def. de } h_{\underline{Nat}}$$

$$= SUC_A(n) \quad \text{, pela def. de } SUC_A$$

$$= SUC_A\left[h_{\underline{Nat}}(SUC^n(Z))\right]$$

$$= SUC_A\left[h_{\underline{Nat}}(t)\right]$$

ii) $h_{\underline{Nat}}\left[PRED_C(t)\right] = h_{\underline{Nat}}\left[PRED_C(SUC^n(Z))\right]$

. No caso em que $n \geq 1$

$$= h_{\underline{Nat}}\left[SUC^{n-1}(Z)\right] \quad \text{, pela def. de } PRED_C$$

$$= n-1 \quad \text{, pela def. de } h_{\underline{Nat}}$$

$$= PRED_A(n) \quad \text{, pela def. de } PRED_A$$

$$= PRED_A\left[h_{\underline{Nat}}(SUC^n(Z))\right]$$

$$= PRED_A\left[h_{\underline{Nat}}(x)\right]$$

. No caso em que $n = 0$ é semelhante

Sejam $t_1 = SUC^m(Z)$ e $t_2 = SUC^n(Z)$, termos de $C_{\underline{Nat}}$.

iii) $h_{\underline{Bool}}\left[MNOR_C(t_1, t_2)\right] = h_{\underline{Bool}}\left[MNOR_C(SUC^m(Z), SUC^n(Z))\right]$

. No caso em que $m \geq n$

$= h_{\underline{Bool}}\ [F]$, pela def. de $MNOR_C$

$= \underline{Falso}$, pela def. de $h_{\underline{Bool}}$

$= MNOR_A(m,n)$, pela def. de $MNOR_A$

$= MNOR_A(h_{\underline{Nat}}(SUC^m(Z)),\ h_{\underline{Nat}}(SUC^n(Z))$

$= MNOR_A(h_{\underline{Nat}}(t_1)\ ,\ h_{Nat}(t_2))$

. No caso em que $m < n$ é semelhante.

b) h é <u>sobrejetora</u> e <u>injetora</u>

<u>Prova:</u>

i) Para $h_{\underline{Bool}}$ é trivial.

ii) Para $h_{\underline{Nat}}$ também é simples, e segue da indução matemática: Seja M o conjunto imagem de $h_{\underline{Nat}}$, ou seja, $M = \{h_{\underline{Nat}}(t) \mid t \in C_{\underline{Nat}}\}$. Por definição $0 \in M$. Por outro lado, como h é um homomorfismo, se $n \in M$, ou seja, $h_{\underline{Nat}}\left[SUC^n(Z)\right] = n$, então $h_{\underline{Nat}}\left[SUC^{n+1}(Z)\right] = n+1$ o que implica que $(n+1) \in M$. Assim M é o próprio conjunto dos naturais, e portanto $h_{\underline{Nat}}$ é sobrejetora. $h_{\underline{Nat}}$ é também injetora pois se $h_{\underline{Nat}}\ [SUC^m(Z)] = h_{\underline{Nat}}\ [(SUC^n(Z)]$, pela definição de $h_{\underline{Nat}}$, $m = n$, e portanto $SUC^m(Z) = SUC^n(Z)$.

II.5 - Uma Metodologia para Construção de Especificações

Que axiomas escrever e quando parar de escrevê-los, isto é, quando os axiomas escritos já são suficientes para especificar o tipo de dados, são algumas das dificuldades que normalmente ocorrem ao especificador de um determinado tipo. Com o objetivo de amenizar essas dificuldades Veloso e Pequeno([36]) desenvolveram uma metodologia de especificação, a qual apresentaremos nessa seção.

De posse da sintaxe do tipo que se deseja especificar formalmente e de sua semântica, expressa por um modelo A, a metodologia em questão consiste basicamente em se escolher uma forma canônica para o tipo de dados e analisar os efeitos da aplicação de cada operação do tipo sobre sua forma canônica. Explicitando, a metodologia é composta dos seguintes passos:

Passo 1 - Escolher uma forma canônica para o tipo de dados, isto é, para cada $s \in S$ um conjunto C_s de termos tal que todo elemento de A_s tenha um único representante em C_s e se $\sigma(t_1,.....,t_n) \in C_s$, então $t_i \in C_{s_i}$ sendo $\sigma: s_1,...,s_n \to s$ um símbolo de operação do tipo.

Passo 2 - Expressar a semântica de cada operação do tipo em termos de seus representantes canônicos, ou seja, usando os representantes canônicos induzir sobre C operações correspondentes àquelas do tipo.

Passo 3 - Para cada símbolo de operação σ do tipo, escrever axiomas para transformar cada termo

da forma $\sigma(c_1,.....,c_n)$, onde $c_1,c_2,....c_n$ são representantes canônicos, no apropriado representante canônico dado em 2, isto é $\sigma_c[c_1,...,c_n]$.

A justificativa de que a presente metodologia conduz a especificações que são ao mesmo tempo completas e corretas está baseada em resultados de Goguen e Thatcher sobre a álgebra dos termos canônicos (ver apêndice) . Os passos 1 e 2 da metodologia garantem que C é uma álgebra de termos canônicos, isomorfa ao modelo, enquanto o passo 3 garante o isomorfismo entre C e a álgebra T/≡.

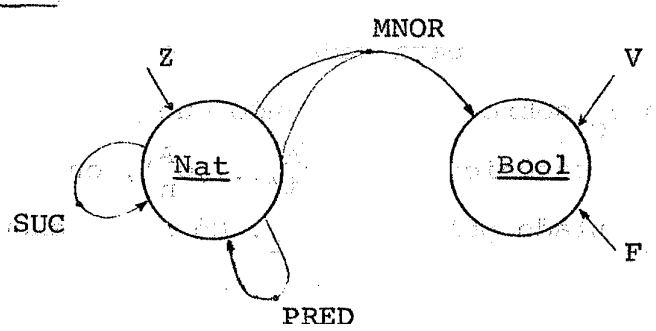
Maiores detalhes são encontrados em Veloso e Pequeno ([36]).

Um Exemplo:

No desenvolvimento da seção II.2, apresentamos uma especificação para os naturais com as operações de sucessor, predecessor, e menor que (álgebra A). Agora, vamos aplicar a metodologia descrita nessa seção e mostrar como aqueles axiomas surgem naturalmente.

Por clareza vamos repetir a sintaxe e a semântica do tipo em questão.

Sintaxe:



Semântica:

A_{Nat} = Conjunto dos naturais ({0,1,2.....})

A_{Bool} = Conjunto de valores lógicos ({Falso,Verdadeiro})

Z_A = 0 (zero)

F_A = Falso

V_A = Verdadeiro

SUC_A = Função sucessor ($SUC_A(n) = n+1$)

$PRED_A$ = Função predecessor ($PRED_A(0) = 0$ e $PRED_A(n) = n-1$, para $n \neq 0$)

$MNOR_A$ = Função lógica < (menor que)

Começaremos por aplicar o passo 1 da metodologia, qual seja, escolher uma forma canônica para o tipo em apreço. Está claro que cada número natural pode ser representado como um número finito de aplicações (talvez zero) da função SUC ao elemento Z, isto é, pelo termo $SUC^n(Z)$, para algum n. Assim, esses termos se constituem numa forma canônica para os naturais. Para os valores lógicos, escolhemos V e F.

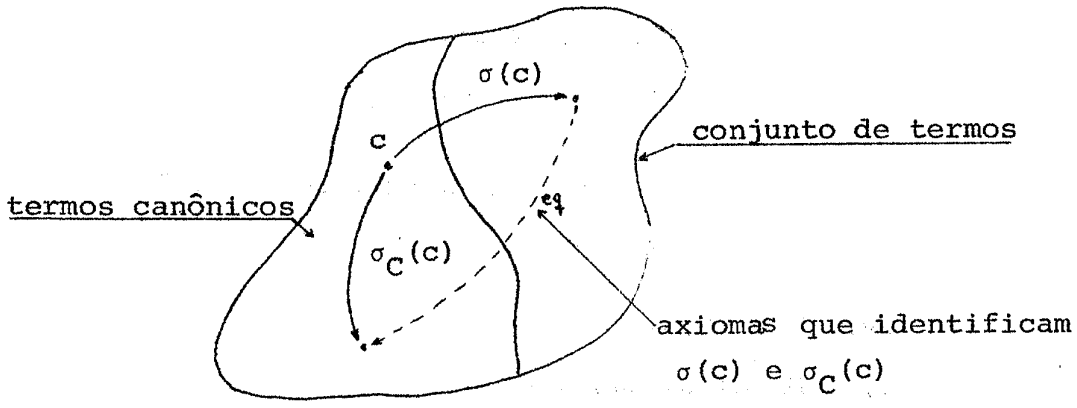
Escolhida uma forma canônica para o tipo (este é o passo mais crítico da metodologia) aplicamos o passo 2, ou seja, descrevemos de forma precisa o efeito de cada operação sobre os termos canônicos. Assim para cada símbolo de operação σ definimos a operação σ_C sobre C estipulando que $\sigma_C(c_1,c_2,\ldots,c_n)$ seja o representante canônico de $\sigma_A(c_i^A,\ldots,c_n^A)$, onde c_i^A é o elemento de A representado pelo termo c_i. No nosso exemplo:

a) $SUC_C \left[SUC^n(Z)\right] = SUC^{n+1}(Z)$

b) $PRED_C \left[SUC^n(Z)\right] = \begin{cases} Z & , \text{Se } n = 0 \\ SUC^{n-1}(Z) & , \text{Se } n \neq 0 \end{cases}$

c) $MNOR_C \left[SUC^m(Z) \ , \ SUC^n(Z)\right] = \begin{cases} F & , \text{Se } m \geq n \\ V & , \text{Se } m < n \end{cases}$

Agora, aplicamos o passo 3 da metodologia, ou seja, procuramos escrever as regras necessárias para transformar os termos da forma $\sigma(c_1, c_2, \ldots, c_n)$ em termos da forma $\sigma_C(c_1, c_2, \ldots c_n)$, obtidos pelo passo 2:



A definição (a) corresponde a uma identidade sintatica, de modo que nenhum axioma se faz necessário.

Para transformar $PRED\left[SUC^n(Z)\right]$ em Z ou $SUC^{n-1}(Z)$, como requer a equação (b), podemos aplicar os seguintes axiomas:

i) $PRED(Z) = Z$

ii) $PRED(SUC(x)) = x$

Obs.: A validade do axioma (ii) pode ser verificada pela substituição de x pelos termos canônicos e utilização das equações (a) e (b).

A transformação de $MNOR(SUC^m(Z),SUC^n(Z))$ em F ou V pode ser feita do seguinte modo. Inicialmente pesquisamos um axioma que reduza simultaneamente o número de SUC's de ambos os argumentos de MNOR. Tal axioma é assim:

iii) $MNOR(SUC(x),SUC(Y)) = MNOR(x,Y)$

A aplicação sucessiva desse último axioma nos conduzirá a algum dos seguintes termos:

$MNOR(Z,Z)$, se $m=n$

$MNOR(SUC^{m-n}(Z),Z)$, se $m>n$

$MNOR(Z,SUC^{n-m}(Z))$, se $m<n$

Para que esses termos sejam reduzidos a F ou V, necessitamos dos seguintes axiomas:

iv) $MNOR(Z,Z) = F$

v) $MNOR(SUC(x),Z) = F$

vi) $MNOR(Z,SUC(x)) = V$

Assim, completamos a especificação dos números naturais com as operações de sucessor, predessessor e menor que. A garantia de que os axiomas apresentados são suficientes para especificar o tipo advém do fato de que qualquer termo pode ser reduzido ao seu representante canônico.

CAPITULO III

IMPLEMENTAÇÃO DE TIPOS ABSTRATOS DE DADOS

III.1 - Introdução

No processo de desenvolvimento de programas que manipulam tipos abstratos de dados é de fundamental importância o conceito de implementação. Uma implementação consiste basicamente na representação de um tipo de dados em outro. O segundo tipo é geralmente um tipo de dados mais concreto que o anterior, no sentido de que ele está, de alguma forma, mais próximo dos tipos de dados da linguagem de programação na qual o programa será expresso. Essa etapa repete-se sucessivamente até que o tipo de dados original seja inteiramente expresso em termos dos tipos de dados da linguagem de programação em apreço.

A noção de implementação tem sido objeto de frequentes discussões e vários esforços foram feitos no sentido de precisar-lhe o significado. Dentre os trabalhos com esse objetivo duas abordagens do problema têm merecido destaque: uma considera a implementação como uma função de abstração, definida do espaço concreto no espaço abstrato; a outra considera uma função de representação que associa a cada termo do tipo a ser implementado um outro do tipo usado na implementação.

A primeira abordagem foi apresentada inicialmente por Hoare em [16] e se constitui numa forma mais natural se o tipo de dado é visto como classes de equivalência dos termos, uma vez que um mesmo objeto abstrato pode ter várias representações.

A outra abordagem é brevemente apresentada por Goguen, Thatcher e Wagner em [10] e de modo mais completo o formal por Ehrig , Kreowski e Padawitz em [04] e [05]. Tal abordagem parece ser mais conveniente no contexto de sistemas de reescrita ou problemas de tradução (Gaudel [09]).

Mais recentemente o problema da implementação foi estudado por Pequeno [32]. Utilizando recursos da lógica matemática ele descreveu formalmente o processo de implementação como uma interpretação entre teorias.

No que se segue vamos tratar da implementação e sua correção seguindo o enfoque devido a Goguen. Uma metodologia para verificação da correção de implementações é exisbida e sua aplicação é ilustrada por meio de um exemplo. Finalmente algumas conclusões serão apontadas.

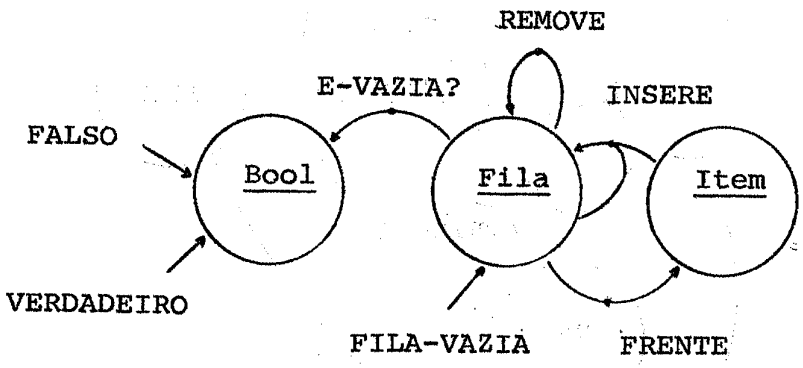
III.2 Implementação

A implementação de um tipo abstrato de dados em outro B (abstrato ou não) é levada a efeito em duas fases: primeiro os objetos de A devem ser representados em função dos objetos de B e, segundo, as operações de A devem ser implementadas através de procedimentos que utilizam as operações de B.

Como exemplo, vamos considerar a implementação do tipo abstrato de dados Fila-de-Item no tipo Vetor-de-Item. Antes porém, apresentaremos as especificações desses tipos, as quais descrevem o comportamento das operações respectivas independentemente de qualquer representação.

A especificação para o tipo Fila-de-Item é a seguinte:

Sintaxe:



Semântica:

Para q: Fila ; it: Item

f1) REMOVE (INSERE(q,it)) = Se E-VAZIA?(q) então FILA-VAZIA
senão INSERE(REMOVE(q),it)

f2) FRENTE(INSERE(q,it)) = Se E-VAZIA?(q) então it
senão FRENTE(q)

f3) E-VAZIA?(FILA-VAZIA) = VERDADEIRO

f4) E-VAZIA?(INSERE(q,it)) = FALSO

Restrições

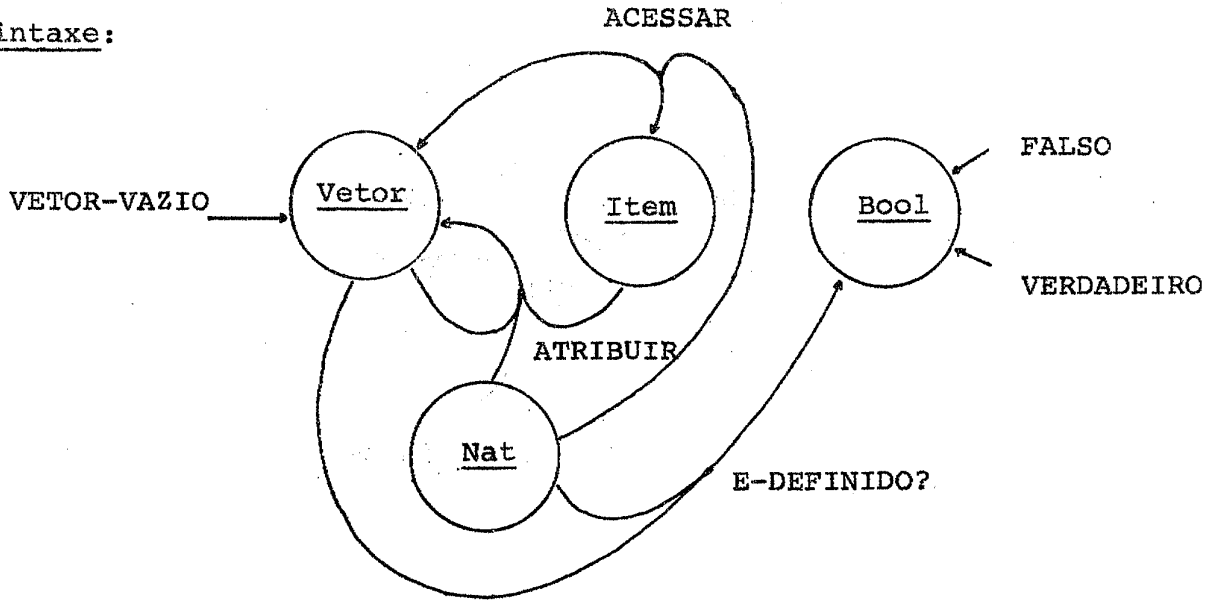
Pre(REMOVE,q) = ¬ E-VAZIA?(q)

Pre(FRENTE,q) = ¬ E-VAZIA?(q)

NOTA: As restrições inseridas na especificação do tipo Fila-de-Item indicam que as operações REMOVE e FRENTE só devem ser aplicadas a filas não vazias. Isto é uma precondição que deve ser verificada pelo usuário do tipo.

A especificação do tipo Vetor-de-Item é a seguinte:

Sintaxe:



Semântica

Para v: Vetor; i,j: Nat;it,it': Item

v1) ATRIBUIR(ATRIBUIR(v,i,it),j,it') = Se i=j
então ATRIBUIR(v,j,it')
senão ATRIBUIR(ATRIBUIR(v,j,it'),i,it)

v2) ACESSAR(ATRIBUIR(v,i,it),j) = Se i=j então it
senão ACESSAR(v,j)

v3) E-DEFINIDO?(VETOR-VAZIO,j) = FALSO

v4) E-DEFINIDO?(ATRIBUIR(v,i,it),j) = Se i=j
então VERDADEIRO
senão E-DEFINIDO?(v,j)

Restrições:

¬ E-DEFINIDO?(v,i) ==> Falha(ACESSAR,v,i)

NOTA i) A restrição na especificação do tipo Vetor-de-Item indica que a aplicação da operação ACESSAR(v,i) falha quando a i-ésima coordenada do vetor v não está definida.

ii) Vamos admitir Nat como sendo os naturais com suas operações e especificação usuais.

Conforme mencionado antes, a implementação de um tipo A em outro B (por exemplo, Fila-de-Item em Vetor-de-Item) consiste na representação dos objetos de A em termos dos objetos de B e na descrição das operações de A em termos das operações de B. De maneira mais formal, a implementação de um tipo de dados A em outro B consiste basicamente em se especificar uma função de representação ρ , que associa a cada sorte s de A um sorte $\rho(s)$ pertencente ao tipo B ou construido a partir dos sortes deste , e a cada operação f de A uma operação f^{ρ} definida a partir das operações de B tal que se $f: s_1 \times s_2 \times \ldots \times s_n \rightarrow s$ então f^{ρ} : $\rho(s_1) \times \rho(s_2) \times \ldots \times \rho(s_n) \rightarrow \rho(s)$. Além disso, ρ deve traduzir na representação a relação de igualdade existente para o tipo a ser implementado.

A função ρ pode ser extendida para os termos do tipo a ser implementado, do modo usual:

$$\rho(f(t_1, t_2, \ldots, t_n)) = f^{\rho}(\rho(t_1), \rho(t_2), \ldots \rho(t_n))$$

Para o exemplo proposto,a implementação do tipo abstrato Fila-de-Item no tipo Vetor-de-Item, escolhemos representar os objetos do sorte Fila através da tripla <Vetor,Nat , Nat>. O primeiro natural da tripla indica a coordenada do vetor onde começa a fila enquanto o segundo aponta a primeira posição livre do vetor, após a fila. Assim desejamos que $\langle [a_0, a_1, \ldots, a_{n-1}], 0, n \rangle$ represente a fila constituida pelos itens $a_0, a_1, \ldots, a_{n-1}$, nesta ordem.

Por simplicidade, limitações sobre o tamanho da fila ou do vetor não serão consideradas.

Assim, para o exemplo em questão, temos:

Função de Representação:

Associação de Sortes:

$\rho(\underline{Bool}) = \underline{Bool}$

$\rho(\underline{Item}) = \underline{Item}$

$\rho(\underline{Fila}) = \underline{Vetor} \times \underline{Nat} \times \underline{Nat}$

Operações Definidas:

Para v,v', :Vetor; i,j,i',j' :Nat; it : Item:

P1) $FALSO^{\rho} = FALSO$

P2) $VERDADEIRO^{\rho} = VERDADEIRO$

P3) $FILA\text{-}VAZIA^{\rho} = \langle VETOR\text{-}VAZIO, 0, 0\rangle$

P4) $INSERE^{\rho}(\langle v,i,j\rangle, it) = \langle ATRIBUIR(v,j,it),i,j+1\rangle$

P5) $REMOVE^{\rho}(\langle v,i,j\rangle) = \langle v,i+1,j\rangle$

P6) $FRENTE^{\rho}(\langle v,i,j\rangle) = ACESSAR(v,i)$

P7) $E\text{-}VAZIA?^{\rho}(\langle v,i,j\rangle) = i \stackrel{?}{=} j$

Tradução da igualdade

r1) $\rho(=_B)$: identidade em Bool

r2) $\rho(=_I)$: identidade em Item

r3) $\langle v,i,j\rangle \; \rho(=_F) \langle v',i',j'\rangle : [(j-i) = (j'-i')] \wedge$

$\wedge \forall k \; [0 \leq k < j-i \Rightarrow ACESSAR(v,i+k) =$

$= ACESSAR(v',i'+k)]$

NOTA: i) B,I e F abreviam Bool, Item e Fila, respectivamente.

ii) Duas triplas de Vetor×Nat×Nat são idênticas se suas componentes o são segundo as especificações correspondentes.

Conforme se depreende da função de representação , os sortes Bool e Item da linguagem de Filas-de-Item foram representados nos próprios na linguagem de Vetor-de-Item, enquanto o sorte Fila foi representado no produto cartesiano Vetor×Nat×Nat.

Cada um dos P1,P2 ....,P7 pode ser visto como especificando um programa que descreve uma operação do tipo Fila-de- Item em termos das operações do tipo Vetor-de-Item. Entretanto , observe que o formalismo usado para se especificar esses programas é semelhante àquele adotado na descrição dos axiomas de uma especificação. procedemos dessa forma para isolar as dificuldades relativas às particularidades e idiossincrasias das diferentes linguagens de programação. Ademais, por não estarem presas a uma linguagem de programação específica, a descrição do processo de implementação e sua correção adquirem um caráter mais geral.

A interpretação da igualdade, especificada por r1, r2 e r3, diz que os sortes Bool e Item têm a mesma representação tanto para Fila-de-Item como para Vetor-de-Item, e que duas triplas, <v,i,j> e <v',i',j'>, representam a mesma fila se as diferenças entre seus subscritos (tamanho da fila) são iguais, e se todos os elementos correspondentes são iguais. Está claro que $\rho(=_B)$, $\rho(=_I)$ e $\rho(=_F)$ são simétricas, reflexivas e transitivas, e dessa forma, relações de equivalência.

III.3. Correção da Implementação

Ficou implícito na escolha da representação do exemplo em exame que nem toda tripla < Vetor, Nat, Nat > representa uma fila. Contudo, para sermos precisos devemos explicitar uma propriedade que caracterize o conjunto das triplas que desejamos tomar como representantes para os objetos do tipo Fila-de-Item. Guttag [14] chama essa propriedade de Invariante da Representação. Para o exemplo em questão, temos:

Invariante da Representação (IR):

$$IR(<v,i,j>) \text{ é } (i \leq j) \wedge \forall k(0 \leq k < j \Longrightarrow E\text{-}DEFINIDO?(v,k) = VERDADEIRO) \wedge \forall k(k \geq j \Longrightarrow E\text{-}DEFINIDO?(v,k) = FALSO)$$

A função de representação ρ é, por construção, um homomorfismo que mapeia os termos do tipo A nos termos do tipo B que satisfazem o invariante da representação.

Como metodologia para demonstrar a correção de uma implementação ρ de um tipo A em outro B, apresentamos a seguinte:

Passo 1: Mostrar que o IR é fechado sob as implementações f^{ρ} das operações f do tipo A.

Passo 2: Mostrar que todo termo de IR é gerado ou é idêntico, pelo axiomas de B, a algum

gerado pelas implementações f^{ρ} das operações de A.

Passo 3: Mostrar que a relação de equivalência induzida pelos axiomas de B restrita a IR está contida na tradução da igualdade $\rho(=)$ de A em B.

Passo 4: Mostrar que cada classe de equivalência gerada em A por seus axiomas é mapeada em uma classe de B segundo $\rho(=)$, ou seja, que os axiomas de A são preservados por $\rho(=)$.

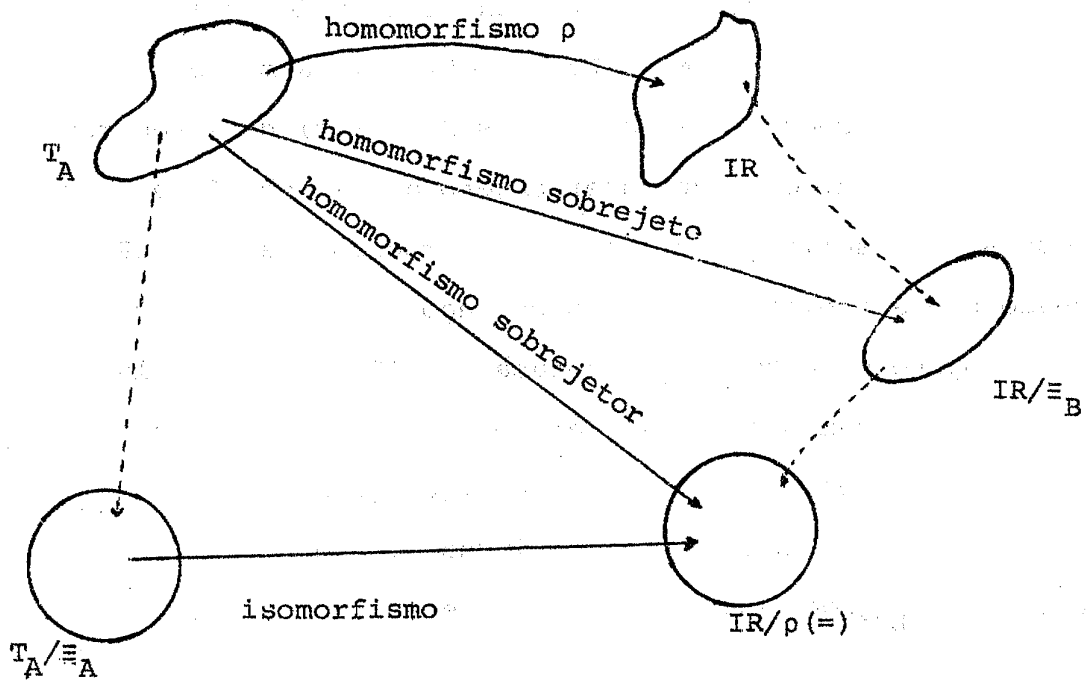
Passo 5: Mostrar que classes de equivalência distintas geradas em A por seus axiomas são mapeadas em classe distintas de B segundo $\rho(=)$, ou seja, $\rho(=)$ está de acordo com os axiomas de A.

A verificação desses passos determina a correção da implementação por garantir a existência de um isomorfismo apropriado. Senão vejamos. Pelo passo 1 da metodologia temos que as constantes (operações nulárias) de A tem representação em IR. Como qualquer objeto de A é obtido a partir destas pela aplicação das outras operações, e como IR é fechado sob a implementação das operações de A, segue-se que todo objeto de A tem representação em IR. Por outro lado, pelo passo 2, temos que qualquer objeto de IR é gerado ou é identificado pelos axiomas de B a algum gerado pela implementação f^{ρ}. Isto significa que ρ induz um homomorfismo sobrejetor em $IR/\equiv_B$ (quociente de IR pela relação de equivalência induzida pelos axiomas de B).

O passo 3 da metodologia diz que $\rho(=)$ está de acordo com os axiomas de B, no sentido de que os objetos de B que são idênticos segundo seus axiomas, também o são segundo $\rho(=)$. Decorre daí que ρ induz um homomorfismo sobrejetor em $IR/\rho(=)$.

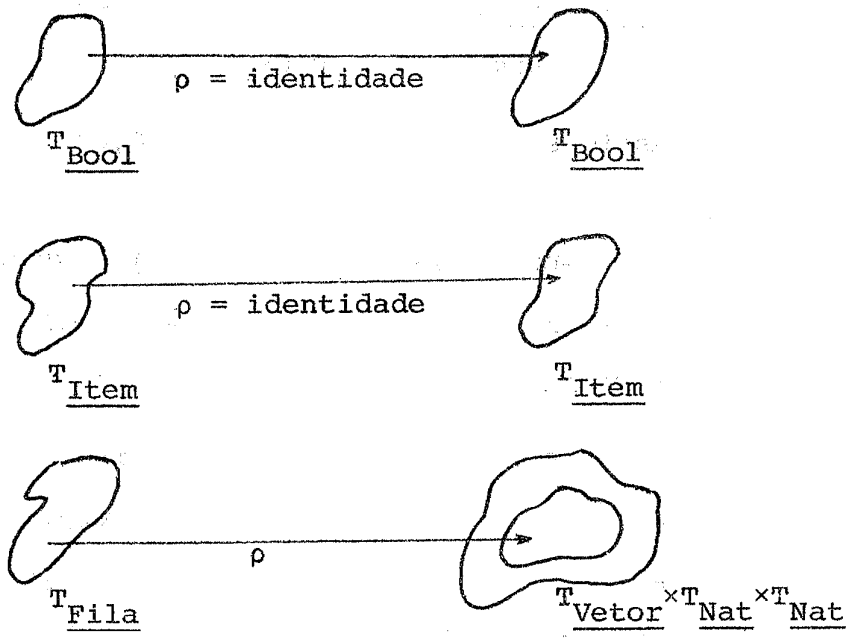
Pelo passo 4 da metodologia, temos que termos de uma mesma classe de equivalência gerada em A por seus axiomas são mapeados em termos de uma mesma classe de equivalência de IR segundo $\rho(=)$, enquanto pelo passo 5, termos pertencentes a classes distintas de $T_A/\equiv_A$ são mapeados em classes distintas de $IR/\rho(=)$. Assim a função de representação ρ induz um isomorfismo de $T_A/\equiv_A$ em $IR/\rho(=)$.

Ilustração Gráfica:



III.4 Aplicação da Metodologia

A função de representação ρ especificada na seção III.2 para o exemplo em análise mapeia os termos dos sortes Bool, Item e Fila nos termos dos sortes Bool, Item e Vetor×Nat×Nat, respectivamente:



É óbvio que a representação identidade de um sorte nele próprio é correta. Desse modo, a verificação de que ρ implementa corretamente o tipo Fila-de-Item no tipo Vetor-de-Item se reduz à verificação de que os termos do sorte Fila são coerentemente representados no espaço dos termos de Vetor×Nat×Nat, isto é, os passos 1,2,3, 4 e 5 da metodologia apresentada na seção anterior se verificam.

No que se segue vamos desenvolver a prova de cada um desses passos.

Passo 1: IR é fechado sob a implementação das operações de Fila-de-Item

Prova: Por indução:

1) Para FILA-VAZIA, temos que FILA-VAZIA$^{\rho}$ =

= <VETOR-VAZIO,0,0>

Assim sendo, IR(<VETOR-VAZIO,0,0>)é $(0 \leq 0) \wedge$

$\wedge \forall k (0 \leq k < 0 \Longrightarrow$

$\Longrightarrow$ E-DEFINIDO?(VETOR-VAZIO,k) = VERDADEIRO)$\wedge$

$\wedge \forall k \ (k \geq 0 \Longrightarrow$ E-DEFINIDO?(VETOR-VAZIO,k) =

= FALSO).

IR(< VETOR-VAZIO, 0,0>) se verifica, pois pelos axiomas dos naturais $0 \leq 0$, $\forall k (0 \leq k < 0 \Longrightarrow$

$\Longrightarrow$ E-DEFINIDO?(VETOR-VAZIO,k) = VERDADEIRO) é válido por vacuidade, e $\forall k (k \geq 0 \Longrightarrow$ E-DEFINIDO? (VETOR VAZIO,K)= FALSO) é válido pelo axioma v3 de Vetor-de-Item. Portanto <VETOR-VAZIO,0,0>$\in$IR.

2) Seja agora uma tripla qualquer <v,i,j> $\in$ IR.

Isso significa que $(i \leq j) \wedge \forall k (0 \leq k < j \Longrightarrow$

$\Longrightarrow$ E-DEFINIDO?(v,k) = VERDADEIRO) $\wedge \forall k (k \geq j \Longrightarrow$

$\Longrightarrow$ E-DEFINIDO?(v,k) = FALSO).

a) Para operação INSERE, temos

INSERE$^{\rho}$(<v,i,j>, it) =

< ATRIBUIR(v,j,it),i,j+1) >

Assim, o IR para tripla resultante é (i≤j+1) ∧
∧ ∀k(0≤k<j+1 ⟹ E-DEFINIDO?(ATRIBUIR(v,j,it),k)=
= VERDADEIRO) ∀k(k≥j+1 ⟹
⟹ E-DEFINIDO?(ATRIBUIR(v,j,it),k)=
= FALSO).

Por hipótese de indução i≤j e assim é verdade que i≤j+1. De igual modo e por v4, E-DEFINIDO?(ATRI - BUIR(v,j,it),k) = VERDADEIRO se verifica para ∀k (o≤k<j).

Isso é válido também para k=j pelo axioma v4 do tipo Vetor-de-Item. Por outro lado, como E-DEFINI DO? (v,k) = FALSO se verifica,para k≥j por hipóte se de indução, por v4 se ve quê E-DEFI - NIDO?(ATRIBUIR(v,j,it),k) = FALSO, para k≥j+1 . Desse modo IR(INSERE$^{\rho}$(<v,i,j> ,it) se verifica, e então INSERE$^{\rho}$(<v,i,j >,it) ∈ IR.

b) Para operação REMOVE, temos REMOVE$^{\rho}$(< v,i,j >) =
= < v,i+1,j >

Assim, IR(< v,i+1,j >) é (i+1≤j)∧ ∀k(0≤k<j ⟹>
⟹> E-DEFINIDO?(v,k) = VERDADEIRO)∧ ∀k(k≥j) ⟹>
⟹ E-DEFINIDO?(v,k) = FALSO).

Por hipótese de indução E-DEFINIDO?(v,k) =
= VERDADEIRO se verifica para 0≤k>j , e de i gual modo E-DEFINIDO?(v,k) = FALSO, para 0k≥j . Também por hipótese de indução i≤j. Porém como

$\langle v,i,j\rangle$ não pode ser FILA-VAZIA$^{\rho}$ pela restrição Pre(REMOVE,q) = ¬E-VAZIA?(q) da especificação do tipo Fila-de-Item, temos que $i<j$ o que implica que $i+1\leq j$. Desse modo, IR($\langle v,i+1,j\rangle$) se verifica e portanto REMOVE$^{\rho}$ ($\langle v,i,j\rangle$) $\in$ IR

□

O que acabamos de mostrar é que a representação de FILA-VAZIA satisfaz o invariante IR, e que se uma fila q tem sua representação $\langle v,i,j\rangle$ satisfazendo IR, a aplicação a q das operações que dão resultados no sorte Fila resulta numa nova q' cuja representação $\langle v',i',j'\rangle$ também satisfaz IR. Como qualquer fila é obtida à partir da FILA-VAZIA pela aplicação das operações INSERE e REMOVE, IR conterá a representação de todas as filas.

Passo 2: Toda tripla de IR é gerada ou é idêntica, pelos axiomas de Vetor×Nat×Nat, a alguma gerada pela implementação das operações de Fila-de-Item.

Prova: Vamos tomar uma tripla qualquer $\langle v,i,j\rangle \in$ IR. Isso significa que $(i\leq j)\wedge \forall k(0\leq k<j \Longrightarrow$ E-DEFINIDO ? (v,k) = VERDADEIRO) $\wedge \forall k(k\geq j \Longrightarrow$ E-DEFINIDO?(v,k) = = FALSO).

Se o vetor v está definido exatamente para $k, 0\leq k<j$, pelos axiomas v1, v3 e v4 de Vetor-de-Item, temos que:

v = ATRIBUIR(...ATRIBUIR(VETOR-VAZIO,k_0,it_{k_0})... ,
,k_{j-1},it_{j-1}), sendo $k_0,k_1,\ldots,k_{j-1}$ uma permutação de 0,1,2,...,j-1.

Consideremos agora

$w = \rho$(REMOVE(...REMOVE(INSERE(...INSERE(FILA-VAZIA,it_0)...,
(i vezes) (j vezes) ,it_{j-1}))...))

Pela definição de ρ temos:

w = REMOVE$^{\rho}$(...REMOVE$^{\rho}$(INSERE$^{\rho}$(...INSERE$^{\rho}$(FILA-VAZIA$^{\rho}$,it_0)...,
, it_{j-1}))...)

Pela aplicação do p3, temos:

w = REMOVE$^{\rho}$(...REMOVE$^{\rho}$(INSERE$^{\rho}$(...INSERE$^{\rho}$(<VETOR-VAZIO,0,0>,
, it_0)...,it_{j-1}))...)

Aplicando INSERE$^{\rho}$ j vezes, temos:

w = REMOVE$^{\rho}$(...REMOVE$^{\rho}$(<ATRIBUIR(...ATRIBUIR(VETOR-VAZIO,
0,it_0)...,j-1,it_{j-1}),0,j>)...)

a aplicação de REMOVE$^{\rho}$ i vezes acarreta:

w = <ATRIBUIR(...ATRIBUIR(VETOR-VAZIO,0,it_0)...j-1,it_{j-1}),i,j>

Agora, pela aplicação dos axiomas de Vetor-de-Item vemos que v é idêntico ao vetor que aparece na tripla w.

Conclusão: Dada uma tripla qualquer <v,i,j> $\in$ IR ela é igual, pelos axiomas das triplas <Vetor,Nat, Nat>, à representação de alguma fila.

□

Pelo passo 1 vimos que para uma fila q qualquer, $q^{\rho} \in IR$. Pelo passo 2 temos que para uma tripla $\langle v,i,j\rangle \in IR$, existe uma fila-de-item q', tal que $(q')^{\rho} \equiv_{V\times N\times N} \langle v,i,j\rangle$ ($\equiv_{V\times N\times N}$ é a relação de equivalência gerada pelos axiomas de Vetor×Nat×Nat em IR). Podemos concluir que a função de representação ρ induz um homomorfismo sobrejetor de T_F em $IR/\equiv_{V\times N\times N}$.

Passo 3: Em IR $\rho(=_F)$ está de acordo com os axiomas de Vetor-de-Item

Prova: Considere duas triplas de IR, $\langle v,i,j\rangle$ e $\langle v',i',j'\rangle$ tais que $\langle v,i,j\rangle \equiv_{V\times N\times N} \langle v',i',j'\rangle$. Devemos mostrar que $\langle v,i,j\rangle \rho(=_F)$ v',i'j'>. Mas $\langle v,i,j\rangle \equiv_{V\times N\times N} \langle v',i',j'\rangle$ significa que $v \equiv_V v'$, $i \equiv_N i'$ e $j \equiv_N j'$, os dois últimos, pela nossa suposição sobre a especificação de Nat, equivalendo a i=i e j=j'. Então, temos $\langle v,i,j\rangle$ e $\langle v', i,j\rangle$ em IR, tais que $v \equiv_V v'$ e queremos mostrar que $\langle v,i,j\rangle \rho(=_F) \langle v',i,j\rangle$. Como $v \equiv_V v'$ se e só se esta igualdade decorre dos axiomas de Vetor-de-Item, basta mostrar que, para cada axioma u=w de Vetor-de-Item, com u e w de sorte Vetor, tem-se $\langle u,i,j\rangle \rho(=_F) \langle w,i,j\rangle$, ou seja, j-i=j-i e para p=i,...,j-1, ACESSAR(u,p)=ACESSAR(w,p). Nestas condições, temos o axioma v1, quando u é

ATRIBUIR(ATRIBUIR(v,k,it), ℓ , it') e w é ATRIBUIR(v,k,it'), caso k=ℓ , ou ATRIBUIR(ATRIBUIR(v,ℓ,it'),k,it) caso contrário. Usando o axioma v2, obtemos.

	ACESSAR(u,p)	ACESSAR(w,p)	
p=ℓ	it'	it'	p=ℓ k=ℓ
		it'	p=ℓ k≠ℓ
p≠ℓ p=k	it	it	p=k k≠ℓ
p≠ℓ p≠k	ACESSAR(v,p)	ACESSAR(v,p)	p=ℓ p≠k
		ACESSAR(v,p)	p=k p≠ℓ k≠ℓ

□

Passo 4: Os axiomas de Fila-de-Item são preservados pela representação

Prova: Considere uma fila q, com representação q^{ρ}=<v,i,j>

a) Seja o axioma f1: REMOVE(INSERE(q,it)) =

Se E-VAZIA?(q)
então FILA-VAZIA
senão INSERE(REMOVE(q),it).

Para o lado esquerdo do axioma, temos:

ρ(REMOVE(INSERE(q,it)) =

= $REMOVE^{\rho}(INSERE^{\rho}(q^{\rho},it))$, pela def. de ρ

= $REMOVE^{\rho}(INSERE^{\rho}(<v,i,j>,it)$

= $REMOVE^{\rho}$(<ATRIBUIR(v,j,it),i,j+1>), por P4

= <ATRIBUIR(v,j,it),i+1,j+1> , por P6

Para o lado direito, temos:

Se ρ(E-VAZIA?(q)) então ρ(FILA-VAZIA)
senão ρ(INSERE(REMOVE(q),it) =

= Se $E\text{-VAZIA?}^{\rho}(q^{\rho})$ então $FILA\text{-VAZIA}^{\rho}$
senão $INSERE^{\rho}(REMOVE^{\rho}(q^{\rho}),it)$

= Se $i \stackrel{?}{\neq} j$ então <VETOR-VAZIO,0 , 0 >
senão $INSERE^{\rho}$ (<v,i+1,j>,it), por p7,p3 e p5

= Se $i \stackrel{?}{=} j$ então <VETOR-VAZIO,0,0 >
senão <ATRIBUIR(v,j,it),i+1,j+1> por P4

Temos dois casos a analisar:

i) Caso i=j

Lado esquerdo: <ATRIBUIR(v,j,it),i+1,j+1>

Lado direito : < VETOR-VAZIO,0 ,0 >

Estes dois termos sao iguais segundo $\rho(=_F)$.

ii) Caso $i \neq j$

Lado esquerdo: < ATRIBUIR(v,j,it),i+1,j+1>

Lado direito : < ATRIBUIR(v,j,it),i+1,j+1 >

Estes dois termos são idênticos, logo iguais segundo $\rho(=_F)$.

Com isso concluímos que o axioma f1 é preservado.

b) Seja o axioma f2: FRENTE(INSERE(q,it)) =

= Se E-VAZIA?(q)

então it senão FRENTE(q)

Para o lado esquerdo do axioma, temos:

ρ(FRENTE(INSERE(q,it)) =

= $FRENTE^{\rho}(INSERE^{\rho}(q^{\rho},it))$, pela def. de ρ

= $FRENTE^{\rho}(INSERE^{\rho}(< v,i,j>, it)$

= $FRENTE^{\rho}(<$ ATRIBUIR(v,j,it),i,j+1>), por P 4

= ACESSAR(ATRIBUIR(v,j,it),i) , por P 6

Para o lado direito, temos , por p7 e p6:

Se ρ(E-VAZIA?(q)) então it senão ρ(FRENTE(q)) =

= Se $E\text{-VAZIA?}^{\rho}(q^{\rho})$ então it senão $FRENTE^{\rho}(q^{\rho})$

= Se $i \stackrel{?}{=} j$ então it senão ACESSAR(v,i)

Temos dois casos a analisar:

i) Caso i=j

Lado esquerdo: ACESSAR(ATRIBUIR(v,j,it),i)=

= it , por v2

Lado direito: it

Os dois termos são idênticos segundo $\rho(=_I)$.

ii) Caso i≠j

Lado esquerdo: ACESSAR(ATRIBUIR(v,j,it),i)=

= ACESSAR(v,i) , por v_2

Lado direito: ACESSAR(v,i)

Estes termos são idênticos segundo $\rho(=_I)$

Com isso concluímos que o axioma f2 é preservado.

c) Seja o axioma f3: E-VAZIA?(FILA-VAZIA) =

= VERDADEIRO

Para o lado esquerdo, temos:

ρ(E-VAZIA?(FILA-VAZIA)) =

= E-VAZIA?$^{\rho}$(FILA-VAZIA$^{\rho}$) , pela def. de ρ

= E-VAZIA?$^{\rho}$(<VETOR-VAZIO,0,ρ>), por P3

= 0 $\stackrel{?}{=}$ 0 , por P7

Para o lado direito, temos, por p2:

ρ(VERDADEIRO)=VERDADEIRO$^{\rho}$= VERDADEIRO.

O axioma se verifica, uma vez que $0 \stackrel{?}{=} 0$ é idêntico a VERDADEIRO, segundo $\rho(=_B)$.

NOTA: Estamos supondo uma especificação usual dos dos naturais.

d) Seja o axioma f4: E-VAZIA?(INSERE(q,it)) = FALSO

Para o lado esquerdo, temos:

ρ(E-VAZIA?(INSERE(q,it)) =

= E-VAZIA?$^{\rho}$(INSERE$^{\rho}$(q$^{\rho}$,it)) , pela def. de ρ

= E-VAZIA?$^{\rho}$(INSERE$^{\rho}$(<v,i,j> ,it))

= E-VAZIA?$^{\rho}$(<ATRIBUIR(v,j,it),i,j+1>) , por P4

= $i \stackrel{?}{=} j+1$, por P7

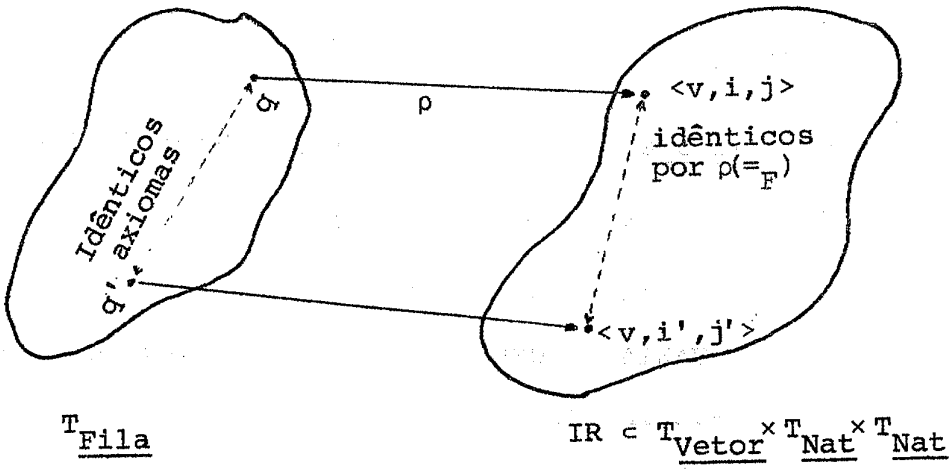
Para o lado direito, temos., por p1:

ρ(FALSO) = FALSO$^{\rho}$ = FALSO

O axioma se verifica, uma vez que $q^{\rho} \in$ IR e portanto $i \leq j$. Assim $i \stackrel{?}{=} j+1$ e idêntico a FALSO segundo $\rho(=_B)$.

□

O que acabamos de demonstrar, utilizando para isto os axiomas de Vetor-de-Item, os axiomas dos naturais e a tradução das igualdades $\rho(=_F)$, $\rho(=_I)$ e $\rho(=_B)$, é que se os axiomas de Fila-de-Item identificam dois termos q e q' então q^{ρ} e $(q')^{\rho}$ são identificados em Vetor×Nat×Nat:



Passo 5: $\rho(=_F)$ está de acordo com os axiomas de Fila-de-Item

Prova: Consideremos duas filas q_1 e q_2 cujas representações sejam identificadas por $\rho(=_F)$, isto é:

$$q_1^\rho \quad \rho(=_F) \quad q_2^\rho$$

Sejam $\bar{q}_1$ e $\bar{q}_2$ representantes canônicos das classes de equivalência que contém q_1 e q_2, respectivamente. Portanto, $q_1 \equiv_F \bar{q}_1$, $q_2 \equiv_F \bar{q}_2$.

Assim, pelo demonstrado no passo 4, ou seja, filas idênticas têm representações idênticas, $q_1^\rho \rho(=_F) \bar{q}_1^\rho$ e $q_2^\rho \ \rho(=_F) \ \bar{q}_2^\rho$, donde se conclui que $\bar{q}_1^\rho \ \rho(=_F) \ \bar{q}_2^\rho$.

Resumindo:

$$\text{por escolha} \left\{ \begin{array}{lllll} q_1 & q_2 & q_1^{\rho} & p(=_F) & q_2^{\rho} \\ \equiv_F & \equiv_F & \rho(=_F) & & \rho(=_F) \\ \bar{q}_1 & \bar{q}_2 & \bar{q}_1^{\rho} & \rho(=_F) & \bar{q}_2 \end{array} \right\} \text{pelo passo 4}$$

Os termos canônicos de Fila-de-Item são da forma:

$$\bar{q}_1 = \text{INSERE}(\text{INSERE}\ldots\text{INSERE}(\text{FILA-VAZIA}, a_0),\ldots,a_n)$$

$$\bar{q}_2 = \text{INSERE}(\text{INSERE}\ldots\text{INSERE}(\text{FILA-VAZIA}, b_0),\ldots,b_m)$$

Pela definição de ρ, temos

$$\rho(\bar{q}_1) = <\text{ATRIBUIR}(\ldots(\text{ATRIBUIR}(\text{ATRIBUIR}(\text{VETOR-VAZIO},0,a_0), \ldots, 1, a_1)\ldots, n, a_n), 0, n>$$

$$\rho(\bar{q}_2) = <\text{ATRIBUIR}(\ldots(\text{ATRIBUIR}(\text{ATRIBUIR}(\text{VETOR-VAZIO},0,b_0), \ldots, 1, b_1)\ldots, m, b_m), 0, m>$$

Pela definição de $\rho(=_F)$, se $\bar{q}_1\ \rho(=_F)\ \bar{q}_2^{\rho}$, necessariamente temos que

$$(n=m) \wedge \forall k\,(0 \leq k < n \Longrightarrow \text{ACESSAR}(v,k) = \text{ACESSAR}(w,k)),$$

onde $v = \text{ATRIBUIR}(\ldots(\text{ATRIBUIR}(\text{ATRIBUIR}(\text{VETOR-VAZIO},0,a_0), \ldots, 1, a_1)\ldots, n, a_n)$

$$w = \text{ATRIBUIR}(\ldots(\text{ATRIBUIR}(\text{ATRIBUIR}(\text{VETOR-VAZIO},0,b_0), \ldots, 1, b_1)\ldots, m, b_m)$$

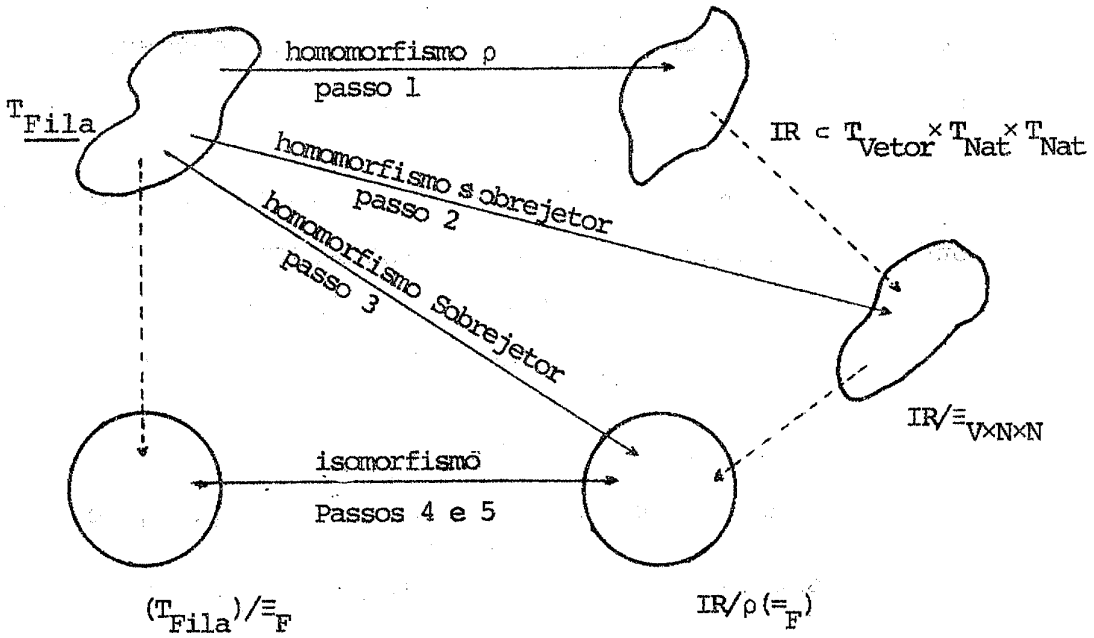
Com $m=n$ e $a_k=b_k$, para $k=0,1,\ldots,n$, temos que $\bar{q}_1$ e $\bar{q}_2$ são a mesma fila e assim $\bar{q}_1 \equiv_F \bar{q}_2$, e portanto $q_1 \equiv_F q_2$.

Conclusão: $\rho(=_F)$ não identifica termos que não sejam representações de filas identificadas por $\equiv_F$

□

Como já observamos na seção III.3, esses cinco passos garantem a existência de um isomorfismo entre $(T_{Fila})/\equiv_F$ e $IR/\rho(=_F)$, e desse modo, que a implementação do tipo Fila-de-item em Vetor-de-Item é correta.

Resumindo graficamente, o efeito de cada passo nessa demonstração, temos:



III.5 Considerações Adicionais

Neste capítulo tratamos da implementação de tipos abstratos de dados especificados algebricamente. Em particular apresentamos e justificamos uma metodologia para verificar a correção de implementação e a ilustramos por meio de um exemplo.

Tentamos, na metodologia apresentada, sistematizar os vários passos da verificação de maneira mais ou menos modular em relação ao que se usa em cada passo. Assim é que no passo 1 são usados apenas os axiomas do tipo B e as precondições do tipo A, além da definição do invariante. Por outro lado, os axiomas de A somente são usados no passo 5, juntamente com os de B e a definição de $\rho(=)$.

Outra observação pertinente se refere às exigências dos vários passos. Por exemplo, no passo 2 da metodologia apresentada, exigimos que os termos de IR sejam gerados ou identificados pelos axiomas de B a algum termo gerado pela implementação, das operações de A. Isso, contudo, não é estritamente necessário. O que realmente importa é cada uma das classes de equivalência de $IR/\rho(=)$ tenha pelos menos um termo gerado pela implementação das operações de A. Por exemplo fácil verificar que a demonstração apresentada na seção III.4 não seria muito diferente se o IR escolhido fosse um pouco mais amplo, como o seguinte:

$$(i \leq j) \wedge \forall k (i \leq k < j) \Longrightarrow \text{E-DEFINIDO?}(v,k) = \text{VERDADEIRO})$$

que deixa em aberto E-DEFINIDO?(v,k) para k fora do intervalo $[i,j-1]$.

Finalmente devemos ressaltar que esta metodologia foi desenvolvida tendo em vista a conceito de implementação baseada em função de representação. Não parece muito dificil adaptá-la para o caso de implementação baseada em função de abstração. Porém é importante ter-se em mente que tudo o que foi dito se refere a tipos abstratos de dados vistos como álgebras iniciais.

CAPÍTULO IV

OUTRAS CONSIDERAÇÕES

Conforme foi enfatizado na introdução, o método de desenvolvimento de programas por refinamentos sucessivos estimula o programador a considerar os aspectos da representação de dados somente após a elaboração de um algoritmo que resolve o problema. Este algoritmo naturalmente faz uso de abstrações, e dessa forma é visto como um programa abstrato operando sobre dados abstratos. Certo da correção do programa abstrato, então, o programador volta sua atenção para a implementação das abstrações , que, por sua vez, pode ser feita em termos de outras abstrações. O processo tem um fim, quando todas as abstrações forem realizadas na linguagem de programação usada para codificação do programa.

O presente trabalho tratou de dois aspectos importantes da metodologia acima mencionada: Especificação e Implementação . Contudo, o tratamento dado a estes aspectos não dá muita ênfase a programação em si.

A especificação tem o papel fundamental de estabelecer um elo de comunicação entre o programa abstrato e a implementação. É usando a especificação, como hipótese, que o programador verifica a correção formal do programa abstrato. A exeqüibilidade dessa hipótese é garantida pela prova de correção da implemen

tação. Estas duas tarefas constituem a prova de correção do programa como um todo.

A implementação, como abordado aqui neste trabalho, não faz referência a nenhuma linguagem de programação. Implementar em uma linguagem de programação, contudo, significa representar os sortes abstratos por meio de sortes construidos a partir dos tipos primitivos da linguagem e definir as operações por meio de procedimentos. Várias linguagens fornecem mecanismos para esse fim. Algumas, como CLU, fornecem um mecanismo de propósito mais geral, o "cluster". Este mecanismo reune num só lugar as informações relevantes da implementação, qual seja, a representação dos sortes e os procedimentos que modelam as operações. Isto constitui um bloco independente fora do qual o tipo é visto abstratamente, pois os detalhes de implementação ficam invisíveis para o programador abstrato. A tarefa do implementador é escrever o "cluster".

Em (34) tratamos dos mecanismos de estruturação de dados em linguagem de programação e da programação com tipos abstratos de dados, complementando a exposição feita aqui.

APÊNDICE

ASPECTOS TEÓRICOS DA ESPECIFICAÇÃO DE TIPOS DE ABSTRATOS DE DADOS

1. Introdução

No desenvolvimento do capítulo II lançamos mão da persuação ao invés da demonstração. Isto se justifica pelo compromisso inicial de produzirmos um texto que servisse apenas de introdução à especificação e implementação de tipos abstratos de dados. Assim sendo, o mencionado capítulo foi escrito segundo uma visão metodológica, deixando-se de fora as construções teóricas (lemas, teoremas, etc.) que justificam os resultados apresentados. Por exemplo, em nenhum momento nos referimos a um tipo abstrato como sendo a classe de isomorfismo de uma álgebra inicial e muito menos que a álgebra dos termos e a álgebra quociente são iniciais na categoria de todas as Σ-álgebras e Σ-ξ-álgebras, respectivamente. Estes resultados são enunciados e demonstrados neste apêndice.

2. Tipos de Dados como Álgebras

Uma álgebra heterogênea ou poli-sortida A é essencialmente uma família de conjuntos A_s, chamados os domínios da álgebra, com uma coleção de operações (funções) entre estes. O conjunto S de índices para os domínios da álgebra é chamado de conjunto de sortes. Como exemplo de uma tal álgebra, seja a seguinte álgebra A:

Sortes:

$S = \{Nat, Bool\}$

Domínios:

$A_{Nat} = \{0,1,2,3,.....\}$

$A_{Bool} = \{f , v\}$

Operações:

Símbolos:

$\Sigma = \{F , V, OU, E, \neg, Z, SUC, +, EQ\}$

Semântica:

F_A : $\longmapsto A_{Bool}$

: $\longmapsto f$

V_A : $\longmapsto A_{Bool}$

: $\longmapsto v$

OU_A : $A_{Bool} \times A_{Bool} \longrightarrow A_{Bool}$

: $\langle f,f \rangle \longmapsto f$

: qualquer outro par $\longmapsto v$

E_A : $A_{Bool} \times A_{Bool} \longrightarrow A_{Bool}$

: $\langle v,v \rangle \longmapsto v$

: qualquer outro par $\longmapsto f$

$\neg_A$: $A_{Bool} \longrightarrow A_{Bool}$

: $v \longmapsto f$

: $f \longmapsto v$

$$Z_A : \longrightarrow A_{\underline{Nat}}$$

$$: \longmapsto 0$$

$$SUC_A : A_{\underline{Nat}} \longrightarrow A_{\underline{Nat}}$$

$$: \langle n \rangle \longmapsto n+1$$

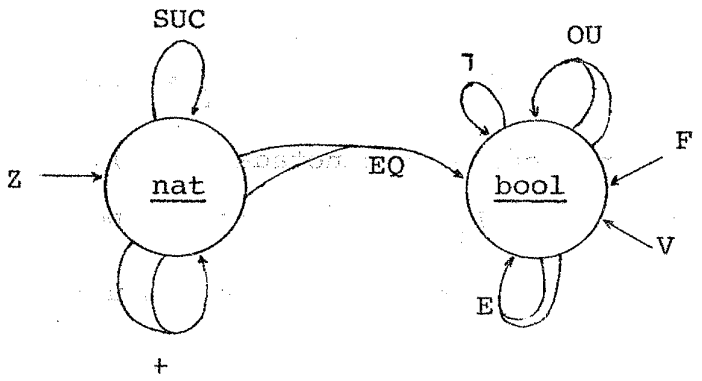
$$+_A : A_{\underline{Nat}} \times A_{\underline{Nat}} \longrightarrow A_{\underline{Nat}}$$

$$: \langle m,n \rangle \longmapsto m+n$$

$$EQ_A : A_{\underline{Nat}} \times A_{\underline{Nat}} \longrightarrow A_{\underline{Bool}}$$

$$: \langle m,n \rangle \longmapsto \begin{cases} v, \text{ se } m = n \\ f, \text{ caso contrário} \end{cases}$$

Usando a notação de Goguen, introduzida no capítulo II, a sintáxe da álgebra acima seria assim expressa:



Vamos definir formalmente o que é uma <u>álgebra S-sortida</u> para um conjunto de sortes S dado

<u>Definição 1</u>. <u>Um domínio de operadores</u> Σ para S é uma família $\Sigma_{w,s}$ de conjuntos, com $s \in S$ e $w \in S^*$ (S^* é o conjunto de todas as cadeias

finitas de símbolos obtidas de S, inclusive a cadeia vazia λ). Se $F \in \Sigma_{w,s}$, dizemos que F é um <u>símbolo operacional</u> de <u>aridade w</u> e de <u>sorte s</u>.

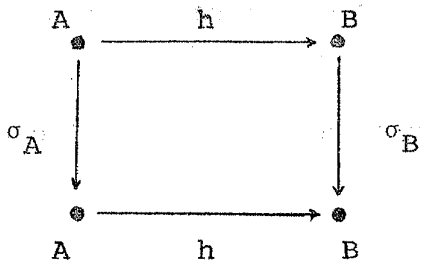
Para o exemplo antes apresentado, temos: $\Sigma_{\lambda,\underline{Bool}}=\{V,F\}$, $\Sigma_{\lambda,\underline{Nat}} = \{Z\}$, $\Sigma_{\underline{Bool},\underline{Bool}} = \{\neg\}$, $\Sigma_{\underline{Bool}\ \underline{Bool},\underline{Bool}} = \{OU,E\}$, $\Sigma_{\underline{Nat},\underline{Nat}} = \{SUC\}$, $\Sigma_{\underline{Nat}\ \underline{Nat},\underline{Nat}} = \{+\}$, $\Sigma_{\underline{Nat}\ \underline{Nat},\underline{Bool}} = \{EQ\}$ e para quaisquer outros w e s, $\Sigma_{w,s}$ é vazio.

<u>Definição 2</u>. Seja Σ um domínio de operadores para S. Uma <u>Σ-álgebra A</u> consiste de um conjunto $A_s \neq \phi$ para cada $s \in S$ (chamado o <u>domínio de A de sorte s</u>) e uma função

$$\sigma_A : A_{s_1} \times A_{s_2} \times \ldots\ldots \times A_{s_n} \longrightarrow A_s$$

para cada $\sigma \in \Sigma_{w,s}$, com $w=s_1\ s_2\ s_3 \ldots s_n$ (chamado a <u>operação de A</u> denotada por σ). Para $\sigma \in \Sigma_{\lambda,s}$, $\sigma_A \in A_s$ (também denotado por $\sigma_A: \longrightarrow A_s$), isto é, $\Sigma_{\lambda,s}$ é o conjunto de <u>nomes das constantes</u> de A para o sorte s.

Por uma função $h:A \longrightarrow B$ entre duas Σ-álgebras A e B queremos indicar uma família de funções $\langle h_s: A_s \longrightarrow B_s \rangle$ para cada $s \in S$. Uma classe de funções de particular interesse aqui, são os <u>morfismos</u>. Tais funções têm a propriedade interessante de preservarem cada operação do conjunto Σ, no seguinte sentido:



$h \circ \sigma_A = \sigma_B \circ h$

Definição 3. Se A e B são duas Σ-álgebras, um Σ-homomorfismo $h:A \longrightarrow B$ é uma família de funções $\langle h_s : A_s \longrightarrow B_s \rangle$, $s \in S$, tal que as operações são preservadas, isto é:

h0) Se $\sigma \in \Sigma_{\lambda,s}$, então $h_s(\sigma_A) = \sigma_B$

h1) Se $\sigma \in \Sigma_{s_1 s_2 \ldots s_n, s}$ e $\langle a_1, a_2, \ldots a_n \rangle \in A_{s_1} \times A_{s_2} \times \ldots A_{s_n}$, então

$$h_s(\sigma_A(a_1, a_2, \ldots, a_n)) =$$

$$= \sigma_B(h_{s_1}(a_1), h_{s_2}(a_2), \ldots, h_{s_n}(a_n)).$$

A composição de homomorfismos é também um homomorfismo . Para cada Σ-álgebra A, a identidade i_A é um Σ-homomorfismo. Um homomorfismo $h:A \longrightarrow B$ é um isomorfismo se e somente se h é injetivo e sobrejetivo.

Definição 4. Uma categoria C de Σ-álgebras consiste de uma classe $|C|$ de Σ-álgebras, as quais

são chamadas os objetos de C, junto com todos os Σ-homomorfismos entre as álgebras.

Introduziremos, agora, o conceito de inicialidade que é fundamental para esta visão de tipos de dados como álgebras.

Definição 5. Uma álgebra A é inicial numa categoria C de Σ-álgebras se e somente se A $\in$ C e para toda álgebra B em C, existe um único homomorfismo h:A ———>B.

Proposição 1

i) Se A e B são álgebras iniciais numa categoria C de Σ-álgebras, então A e B são isomorfas.

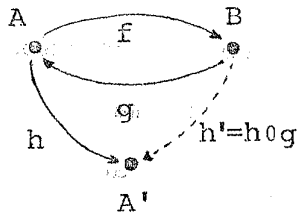
ii) Se uma álgebra B em C é isomorfa a uma álgebra A que é inicial em C, então B também é inicial.

Prova:

i) Como A e B são ambas iniciais, seja h:A ———>B o único homomorfismo de A em B e g:B ———>A o único de B em A. A composição $g^{0}h$:A ———>A é um homomorfismo de A em A, assim como a identidade i_A:A ———>A. Porém como A é inicial, pela unicidade do homomorfismo, temos que $g^{0}h = i_A$. De modo similar, $h^{0}g = i_B$. Assim, h é sobrejetiva e injetiva, e desse modo, um isomorfismo de A em B. Em outras palavras, A e B são isomorfas.

ii) Seja f:A ———>B um isomorfismo com inversa g:B———>A,

sendo A inicial. Seja, agora, A' uma álgebra qualquer de C. Como A é inicial em C, seja h:A ——>A' o único homomorfismo de A em A'. Tomemos h'=h∘g : B——>A' que é um homomorfismo de B em A'.



Provaremos que h' é o único homomorfismo de B em A'. Para isso suponha que h":B ——>A' seja também um homomorfismo de B em A'. Assim, h"∘f:A——>A' é um homomorfismo de A em A' e, como A é inicial, h"∘f = h. Desse modo, h'=h∘g=h"∘f∘g e portanto, como f∘g = i_B, h'=h". Assim o homomorfismo h' entre B e uma álgebra qualquer A' em C é único, donde conclui-se ser B inicial em C. □

É uma prática comum em álgebra abstrata identificar objetos isomorfos, isto é, tratá-los como idênticos. Desse modo, ao falarmos de uma álgebra inicial numa categoria C de Σ-álgebras, na verdade estamos falando de todas aquelas que são isomorfas a esta. O isomorfismo entre duas dessas álgebras é único, como assegura a proposição 1. O fato importante, aqui, é que a classe das álgebras iniciais é caracterizada únicamente sob isomorfismo, isto é, os objetos são caracterizados abstratamente no sentido de que essa caracterização independe das representações, mas é feita somente em termos das estruturas. Isto é o que se deseja para um tipo abstrato, daí a definição seguinte.

Definição 6. Um tipo abstrato de dado é a classe de isomorfismo de uma álgebra inicial numa

categoria C de Σ-álgebras.

É óbvio que, para essa última definição ser de alguma valia, nós precisamos garantir a existência de álgebras iniciais. Além do mais, é nosso desejo saber como esses objetos são. Para tal, vamos construir uma Σ-álgebra S-sortida, a álgebra dos termos T_Σ, cujos sortes são constituidos dos termos gerados por $\Sigma_{w,s}$, $w \in S^+$ e $s \in S$, a partir das constantes $\Sigma_{\lambda,s}$ (Universo de Herbrand).

Definição 7. Seja Σ um domínio de operadores para S. A álgebra dos termos T_Σ consiste de uma Σ-álgebra definida do seguinte modo:

i) Domínios

t0) Se $\sigma \in \Sigma_{\lambda,s}$, então $\sigma \in T_{\Sigma,s}$.

t1) Se $\sigma \in \Sigma_{w,s}$, $w=s_1s_2\ldots s_n$ e $t_i \in T_{\Sigma,s_i}$, então $\sigma(t_1,t_2,\ldots,t_n) \in T_{\Sigma,s}$.

t2) Os únicos elementos de $T_{\Sigma,s}$ são aqueles gerados por t0 e t1.

ii) Operações

op1) Para $\sigma \in \Sigma_{\lambda,s}$, $\sigma_T = \sigma \in T_{\Sigma,s}$

op2) Para $\sigma \in \Sigma_{w,s}$, $w = s_1s_2\ldots s_n$ e $t_i \in T_{\Sigma,s_i}$

$$\sigma_T(t_1,t_2,\ldots,t_n) = \sigma(t_1,t_2,\ldots t_n) \in T_{\Sigma,s}$$

Observe que os elementos dos domínios $T_{\Sigma,s}$ são construidos recursivamente a partir das constantes $\sigma\epsilon\Sigma_{\lambda,s}$, inicialmente colocadas em $T_{\Sigma,s}$, ao se colocar o símbolo de operação $\sigma\epsilon\Sigma_{w,s}$, $w \neq \lambda$, na frente dos termos que são argumentos da operação (os parênteses são usados apenas com o intuito de auxiliar o entendimento). As operações σ_T têm definições apropriadas para gerarem os domínios.

Teorema 2

T_Σ é uma álgebra inicial na categoria Alg_Σ de todas as Σ-álgebras.

Prova:

Para provar a inicialidade de T_Σ vamos proceder do seguinte modo. Primeiro, vamos definir uma família W de conjuntos, a qual daremos uma estrutura de Σ-álgebra e provaremos ser inicial. Segundo, mostraremos que W é igual a T_Σ.

A idéia é construirmos W como uma união disjunta de famílias de conjuntos W^k de Σ-termos de profundidade $k \geq 0$, onde naturalmente $(U_{k\geq 0} W^k)_s = U_{k\geq 0} W^k_s$, sendo W^k_s definido indutivamente, assim:

$$\text{wo)}\quad W^0_s = \Sigma_{\lambda,s}\ , \text{ para } s \in S.$$

$$\text{w1)}\quad W^k_s = \{\sigma(t_1,t_2,\ldots,t_n) \mid n\geq 1,\ \sigma\epsilon\Sigma_{s_1\ldots s_n,s}$$

$$\text{e } t_i \epsilon W^{j_i}_{s_i} \text{ para algum } j_1,j_2,\ldots,j_n < k$$

$$\text{com } \max\{j_1,j_2,\ldots j_n\} = k-1\}.$$

Observe que $W^k \cap W^m = \emptyset$ se $k \neq m$

Agora, vamos tomar $W = U_{k\geq 0} W^k$ e dar a W uma estrutura de Σ-álgebra, como segue:

i) Para $\sigma \in \Sigma_{\lambda,s}$, $\sigma_W = \sigma$ $(\sigma \in W_s^0 \subseteq W_s)$.

ii) Para $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, $\sigma_W(t_1,\ldots,t_n) = \sigma(t_1,\ldots t_n)$

onde, se $t_i \in W_{s_i}^{j_i}$ para $i = 1,2,\ldots,n$, temos

$\sigma_W(t_1,\ldots t_n) \in W_s^k$, onde $k = \max\{j_1,\ldots,j_n\}+1$.

Para mostrar que a Σ-álgebra W é <u>inicial</u>, vamos tomar uma Σ-álgebra B qualquer e construir um Σ-homomorfismo $h:W \longrightarrow B$, parte por parte, definindo $h^k:W^k \longrightarrow B$ e fazendo $h=U_{k\geq 0} h^k$. Note que h e a família h^k determinam únicamente um ao outro, visto que W é uma união disjunta de W^k). Em seguida mostraremos que h é o único Σ-homomorfismo de W em B.

Definamos $h_s^0 : W_s^0 \longrightarrow B_s$ por

$$hw0)\ h_s^0(\sigma) = \sigma_B, \text{ para } \sigma \in \Sigma_{\lambda,s}$$

Agora suponha que $k>0$ e que h^j é definido para $0\leq j<k$; seja $t \in W_s^k$. Então por (w1), t é da forma $\sigma(t_1,t_2,\ldots,t_n), t_i \in W_{s_i}^{j_i}$ para $i = 1,2,\ldots,n$, $n>0$, $\sigma \in \Sigma_{s_1,\ldots s_n,s}$, $k = \max\{j_1,\ldots,j_n\}+1$. Nós definimos então $h_s^k : W_s^k \longrightarrow B_s$ por

hw1) $h_s^k(t) = h_s^k(\sigma(t_1, t_2, \ldots, t_n))$

$$= \sigma_B(h_{s_1}^{j_1}(t_1), h_{s_2}^{j_2}(t_2), \ldots, h_{s_n}^{j_n}(t_n))$$

Com (hw0) e (hw1) temos definido $h^k: W^k \longrightarrow B$ para todo $k \geq 0$ e dessa forma $h: W \longrightarrow B$.

É claro que $h: W \longrightarrow B$ é um Σ-homomorfismo, pois a condição (h^0) da definição 3 decorre naturalmente de (hw0), e como requer (h^1) da mesma definição, todo termo não-constante em W é da forma $\sigma(t_1, t_2, \ldots, t_n) \in W^k$ para algum $k>0$, e desse modo podemos aplicar (hw1) que é exatamente (h1), a menos dos subscritos.

Para mostrar que h é o único Σ-homomorfismo de W em B é suficiente verificar que as condições (h0) e (h1) da definição 3 implicam nas condições (hw0) e (hw1) usadas na definição de h. De fato, (h^0) traduz-se diretamente em (hw0), e (h1) em (hw1), após se acrescentar os subscritos necessários surgidos na definição de W como $U_{k \geq 0} W^k$ e h^k como uma restrição de h para W^k.

Para completar a prova do teorema vamos mostrar que (i) $W \subseteq T_\Sigma$ e que (ii) $T_\Sigma \subseteq W$, e desse modo $T_\Sigma = W$.

i) Por indução:

a) $\Sigma_{\lambda,s} \subseteq T_{\Sigma,s}$, e como $W_s^0 = \Sigma_{\lambda,s}$, então $W^0 \subseteq T_\Sigma$.

b) Suponha que $W^j \subseteq T_\Sigma$ para $0 \leqslant j < k$, e tome $t = \sigma(t_1, \ldots, t_n)$, $t \in W_s^k$. Então $t_i \in W_{s_i}^{j_i}$ com $0 \leqslant j_i < k$, de modo que $t_i \in T_{\Sigma,s_i}$, para $i=1,2,\ldots,n$. Assim, pela condição $(t1_n)$ da definição 7, $t \in T_{\Sigma,s}$ e portanto $W^k \subseteq T_\Sigma$.

ii) Por (t2) temos que $T_{\Sigma,s}$ é o menor conjunto que satisfaz (t0) e (t1). Assim, se mostrarmos que W satisfaz (t0) e (t_1), certamente $T_\Sigma \subseteq W$.

Para (t0): $\Sigma_{\lambda,s} \subseteq W_s$ uma vez que $\Sigma_{\lambda,s} = W_s^0 \subseteq W_s$

Para (t1): Seja $\sigma \in \Sigma_{s_1 s_2 \ldots s_n, s}$ e $t_i \in W_{s_i}$. Então para cada $i=1,2,\ldots,n$, $t_i \in W_{s_i}^{j_i}$ para um único j_i. Seja $k = \max\{j_1,\ldots,j_n\} + 1$. Então $\sigma(t_1,\ldots,t_n) \in W_s^k \subseteq W_s$.

□

Pelo teorema 2 temos a garantia de que, dado um domínio de operadores Σ para S, a Σ-álgebra T_Σ é inicial na categoria Alg_Σ de todas as Σ-álgebras. Isto significa dizer que existe um único homomorfismo de T_Σ em B, onde B é uma Σ-álgebra qualquer. Vale dizer também que as operações de T_Σ têm suas estruturas retratadas na Σ-álgebra B.

Os termos dos sortes de T_Σ foram obtidos a partir das constantes $\Sigma_{\lambda,s}$ pela aplicação a estas das operações $\Sigma_{w,s}$, $w \neq \lambda$, segundo a definição 7. Isto quer dizer que qualquer termo é composto de símbolos de operações (funções) aplicadas a constantes. Pretendemos, agora, fazer uma ligeira modificação, ao introduzir em T_Σ as variáveis. Para isso vamos considerar

uma família disjunta de conjuntos de símbolos, diferentes daqueles de Σ, $X_s = \{x_{s,n} \mid n \in N\}$, $s \in S$. Cada um dos X_s se constitui num conjunto de variáveis para o sorte s.

> Definição 8. Seja Σ um domínio de operadores para S, e $Y = \langle Y_s \rangle$ uma família S-indexada de conjuntos, com $Y_s \subseteq X_s$. $\Sigma(Y)$ é um domínio de operadores obtido do seguinte modo:
>
> i) $\Sigma(Y)_{\lambda,s} = \Sigma_{\lambda,s} \cup Y_s$
>
> ii) $\Sigma(Y)_{w,s} = \Sigma_{w,s}$, com $w \neq \lambda$

O domínio de operadores $\Sigma(Y)$ é obtido de Σ pela introdução neste dos símbolos de variáveis Y, vistos como constantes. Com isso temos uma $\Sigma(Y)$-álgebra ao invés de uma Σ-álgebra. Podemos, contudo, sem prejuízo algum, olhar para os domínios de $T_{\Sigma(Y)}$ como sendo os domínios da Σ-álgebra (Para ressaltar que assim estamos procedendo vamos usar a notação $T_\Sigma(Y)$). Esse fato é importante na proposição 3. Antes, porém, vamos definir o que é uma atribuição de valor a uma variável.

> Definição 9. Seja A uma Σ-álgebra S-sortida e $Y \subseteq X$ uma família S-indexada de conjuntos. Uma atribuição de valor ou interpretação é uma função $\theta: Y \longrightarrow A$ (isto é, $\langle \theta_s : Y_s \longrightarrow A_s \rangle$) que associa a cada variável do sorte s em Y um valor do sorte s em A.

O resultado seguinte é importante porque mostra que pelo fato de termos expandido T_Σ para $T_\Sigma(Y)$, o qual inclui, agora, os termos com variáveis, não afeta substancialmente o conteúdo do teorema 2.

Proposição 3

Seja A uma Σ-álgebra e $\theta : Y \longrightarrow A$ uma atribuição de valor. Então existe um único Σ-homomorfismo $\overline{\theta} : T_\Sigma(Y) \longrightarrow A$ que extende θ no sentido de que $\overline{\theta}_s(x) = \theta_s(x)$ para todo $s \in S$ e $x \in Y_s$.

Prova:

A é uma Σ-álgebra. Dado $\theta : Y \longrightarrow A$, podemos tornar A uma $\Sigma(Y)$-álgebra fazendo x ser o nome do valor de $\theta(x)$ em A ($x_A = \theta(x)$). Pelo teorema 2, então, existe um único $\Sigma(Y)$-homomorfismo $\overline{\theta} : T_\Sigma(Y) \longrightarrow A$, e (pela condição (h0) da definição de homomorfismo) $\overline{\theta}(x_T) = \overline{\theta}(x) = x_A = \theta(x)$, de forma que $\overline{\theta}$ exten de θ. Pelo fato de $\overline{\theta}$ ser um $\Sigma(Y)$-homomorfismo é imediato que ele é um Σ-homomorfismo. Para mostrar a unicidade de $\overline{\theta}$, seja $h : T_\Sigma(Y) \longrightarrow A$ uma Σ-homomorfismo com $h(x) = \theta(x)$. Então h é também um $\Sigma(Y)$-homomorfismo. Como este é único, concluimos que $h = \overline{\theta}$.

□

O Σ-homomorfismo $\overline{\theta} : T_\Sigma(Y) \longrightarrow A$ é uma função que toma um termo e o avalia em A, substituindo as variáveis que ocorrem no termo por seu valor conforme dado por θ.

Definição 10. Uma Σ-equação é um par $e = \langle L,R \rangle$, onde L e R são termos de um mesmo sorte s em $T_\Sigma(Y)$. Notaremos por Var(e) o conjunto das variáveis que ocorrem em e, ou seja, Var(e) = Var(L) U Var(R).

Dado uma equação $e = \langle L,R \rangle$ (usualmente denotada por L = R) e uma atribuição de valores para Var(e), podemos avaliar seu lado direito L e seu lado esquerdo R. Se esses valores são idênticos, dizemos que a atribuição satisfaz a equação.

Definição 11. Seja $e = \langle L,R \rangle$ uma Σ-equação. Dizemos que uma Σ-álgebra A satisfaz e se, e somente se, $\bar{\theta}(L) = \bar{\theta}(R)$ para toda atribuição $\theta: Y \longrightarrow A$, onde Y = Var(e). Se ξ é um conjunto de Σ-equações, então A satisfaz ξ se, e somente se, A satisfaz toda $e \in \xi$. Uma álgebra satisfazendo ξ é dito ser uma Σ-ξ-álgebra.

A categoria de todas Σ-ξ-álgebras, denotada por $Alg_{\Sigma,\xi}$, conforme demonstraremos, também tem uma álgebra inicial a ser denotada por $T_{\Sigma,\xi}$. Nos diremos que $T_{\Sigma,\xi}$, e assim o correspondente tipo abstrato de dado, é representado por ξ.

Definição 12 Uma Σ -congruência $\equiv$ sobre uma Σ-álgebra é uma família $\langle \equiv_s \rangle$, $s \in S$, de relações de equivalência $\equiv_s$ sobre A_s para $s \in S$, tal que para $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, se

$a_i, a'_i \in A_{s_i}$ e se $a_i \equiv_{s_i} a'_i$ para $i = 1, 2, \ldots, n$, então $\sigma_A(a_1, \ldots, a_n) \equiv_s \sigma_A(a'_1, \ldots, a'_n)$.

Se A é uma Σ-álgebra e $\equiv$ uma Σ-congruência sobre A, seja $(A/\equiv)_s = A_s/\equiv_s$ o conjunto das $\equiv_s$-classes de equivalência de A_s. Se $a \in A_s$, seja $[a]_s$ a classe que contém a . Podemos, então , colocar a família S-indexada $A/\equiv$ dentro da categoria das Σ-álgebras, conforme feito a seguir:

Definição 13. Seja A uma Σ-álgebra e $\equiv$ uma Σ-congruência sobre A. A álgebra quociente de A por $\equiv$, denotado por $A/\equiv$, é uma Σ-álgebra definida do seguinte modo:

i) Domínios

$(A/\equiv)_s$, para $s \in S$

ii) Operações

q0) Se $\sigma \in \Sigma_{\lambda,s}$, então $\sigma_{A/\equiv} = [\sigma_A]$

q1) Se $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ e $[a_i] \in (A/\equiv)_{s_i}$ para $i=1,2,\ldots,n$, então $\sigma_{A/\equiv}([a_1],\ldots,[a_n]) = [\sigma_A(a_1,\ldots,a_n)]$.

As operações $\sigma_{A/\equiv}$ da álgebra $A/\equiv$ estão bem definidas, uma vez que a propriedade de congruência garante que o resultado da aplicação de $\sigma_{A/\equiv}$ independe da escolha dos representantes

das classes de equivalência. De fato, se $[a_i] = [a'_i]$ para $i=1,2,\ldots,n$, então $a_i \equiv_{s_i} a'_i$ de modo que $\sigma_A(a_1,\ldots a_n) \equiv_s \sigma_A(a'_1,\ldots a'_n)$, isto é $[\sigma_A(a_1,\ldots a_n)] = [\sigma_A(a'_1,\ldots,a'_n)]$.

A álgebra inicial $T_{\Sigma,\xi}$, na categoria de todas as Σ-álgebras que satisfazem ξ, é obtida ao se tomar o quociente de T_Σ pela relação de congruência gerada por ξ. Antes de mostrarmos esse fato vamos precisar o que seja a congruência gerada por uma relação arbitrária sobre uma álgebra A.

Proposição 4

Seja A uma Σ-álgebra, e seja R uma relação sobre A (isto é, como usual, $R = \langle R_s \rangle, s\in S$, para $R_s \subseteq A_s A_s$). Então existe uma menor Σ-congruência sobre A contendo R, a qual é chamada de relação de congruência gerada por R em A.

Prova:

Seja $\overset{*}{K}(R)$ o conjunto de todas as relações de congruência sobre A que contêm R. $\overset{*}{K}(R) \neq \emptyset$ pois a relação $\Psi = \langle \Psi_s = A_s \times A_s \mid s\in S\rangle$ está em $\overset{*}{K}(R)$. Agora seja $\equiv_R = \bigcap \overset{*}{K}(R)$; isto é, para cada $s\in S, (\equiv_R)_s = \bigcap \{K_s \mid K \in \overset{*}{K}(R)\}$. Então $\equiv_R$ é uma relação de congruência sobre A pois se $\sigma \in \Sigma_{s_1\ldots s_n,s}$ e $a_i, a'_i \in A_{s_i}$ e se $a_i \equiv_R a'_i$, para $i = 1,\ldots,n$, então $a_i \, K \, a'_i$ para todo $K \in \overset{*}{K}(R)$, portanto, pela definição de congruência, $\sigma(a_i,\ldots a_n) \, K \, \sigma(a'_1,\ldots,a'_n)$ para todo $K \in \overset{*}{K}(R)$. Concluimos então que $\sigma(a_1,\ldots a_n) \equiv_R \sigma(a'_1,\ldots,a'_n)$, e então $\equiv_R$ é, de fato, uma congruência.

Uma Σ-representação ξ determina uma relação $\xi(A)$ sobre qualquer Σ-álgebra A na qual $\xi(A)_s$ é o conjunto de todos os pares $\langle\overline{\theta}_s(L), \overline{\theta}_s(R)\rangle$ tal que $e = \langle L,R\rangle$ está em ξ_s e $\theta:Var(e)\longrightarrow A$ é uma atribuição de valor. Assim $\xi(A)_s$ é o conjunto de pares de elementos do domínio A_s que ξ requer que sejam iguais ou identificados.

Teorema 5

Seja ξ uma Σ-representação, e seja $\equiv_\xi$ a Σ-congruência sobre T_Σ gerada pela relação $\xi(T_\Sigma)$. Então $T_\Sigma/\equiv\xi$, o quociente de T_Σ por $\equiv_\xi$ (Notação: $T_{\Sigma,\xi}$), é inicial na categoria $Alg_{\Sigma,\xi}$ de todas as Σ-álgebras satisfazendo ξ.

Antes de apresentarmos a demonstração desse teorema vamos anunciar e provar 3 lemas usados na mesma.

Lema 5A

Se $h:B\longrightarrow A$ é um Σ-homomorfismo, seja $(\equiv_h)_s = \{\langle b,b'\rangle \mid b,b' \in B_s \text{ e } h_s(b) = h_s(b')\}$ para $s\in S$. Então $\equiv_h$ é uma Σ-congruência sobre B. Reciprocamente, se $\equiv$ é uma congruência sobre B, então a atribuição $b\in B \longrightarrow [b] \in B/\equiv$ define um Σ-homomorfismo.

Prova:

Primeiro, cada $(\equiv_h)_s$ é claramente uma relação de equivalência. Agora, seja $\sigma\in\Sigma_{s_1\ldots s_n,s}$ e suponha $b_i(\equiv_h)_{s_i}b'_i$ para

$i = 1,\ldots,n$. Então $h_s(\sigma_B(b_1,\ldots,b_n)) = \sigma_A(h_{s_1}(b_1),\ldots,h_{s_n}(b_n)) =$ $= \sigma_A(h_{s_1}(b'_1),\ldots,h_{s_n}(b'_n)) = h_s(\sigma_B(b'_1,\ldots,b'_n))$, e desse modo $\sigma_B(b_1,\ldots,b_n)\ (\equiv_h)_s\ \sigma_B(b'_1,\ldots,b'_n)$. Para provar a segunda parte do lema, seja $g_s(b) = [b]_s, [b]_s \in (B/\equiv)_s$. Que g é um Σ-homomorfismo decorre diretamente da definição de $B/\equiv$: Para $\sigma\in\Sigma_{s_1\ldots s_n,s}$ e $b_i \in B_{s_i}$, $g_s(\sigma_B(b_1,\ldots,b_n)) = [\sigma_B(b_1,\ldots,b_n)]_s = \sigma_{B/\equiv}([b_1],\ldots[b_n])$ $= \sigma_{B/\equiv}(g_{s_1}(b_1),\ldots,g_{s_n}(b_n))$.

□

Lema 5B

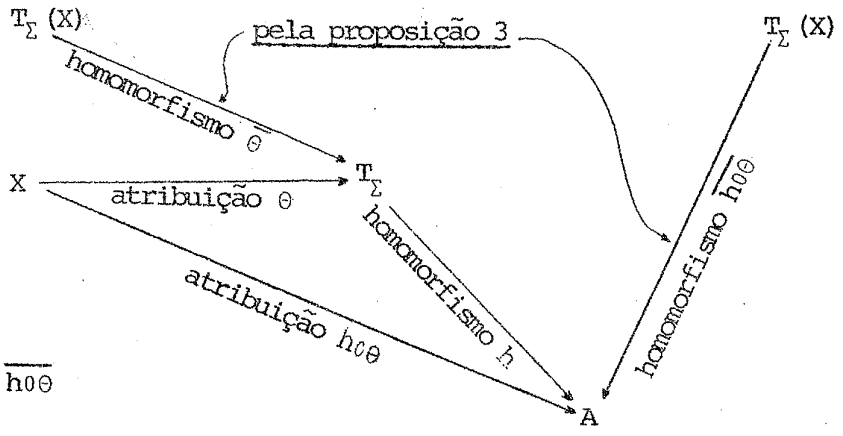
Se $\theta: X \longrightarrow T_\Sigma$ é uma substituição (isto é, uma atribuição de um termo sem variável à uma variável $x \in X$) e $h: T_\Sigma \longrightarrow A$ é o único homomorfismo de T_Σ na Σ-álgebra A, então $h\circ\theta$ é uma atribuição de valor de X em A e $\overline{h\circ\theta} = h\circ\bar{\theta}$.

Prova:

$h\circ\bar{\theta}$ é uma composição de homomorfismos e portanto um homomorfismo. Por outro lado, $h\circ\bar{\theta}(x) = h\circ\theta(x)$, de modo que $\overline{h\circ\theta} = h\circ\bar{\theta}$.

□

O esquema abaixo esclarece melhor o siginificado do lema 5B.



Lema 5C

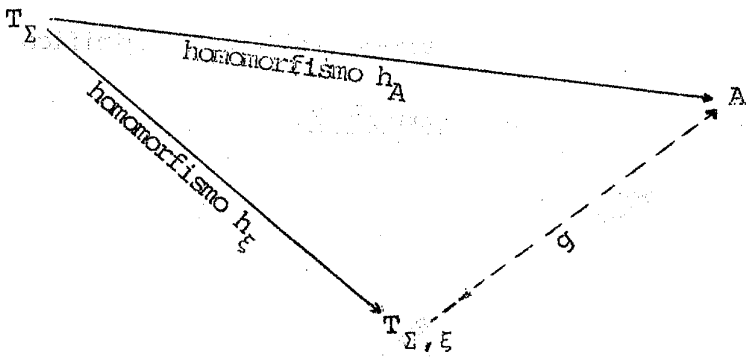
Seja A uma Σ-álgebra satisfazendo um conjunto ξ de Σ-equações, e seja $\equiv_A$ a Σ-congruência sobre T_Σ determinada pelo único homomorfismo $h_A : T_\Sigma \rightarrow A$ via lema 5A. Então $\equiv_\xi \subseteq \equiv_A$.

Prova:

É suficiente mostrar que $\xi(T_\Sigma) \subseteq \equiv_A$, uma vez que $\equiv_\xi$ é a menor congruência sobre T_Σ que contém $\xi(T_\Sigma)$. Seja, então, $< \bar{\theta}(L)$, $\bar{\theta}(R) >$ um par qualquer em $\xi(T_\Sigma)$. Pelo Lema 5B $h \theta \theta(L) = \overline{h \theta \theta}(L)$, assim como $h \theta \bar{\theta}(R) = \overline{h \theta \theta}(R)$. Uma vez que A satisfaz ξ , $\overline{h \theta \theta}(L) = \overline{h \theta \theta}(R)$. Assim, $h(\bar{\theta}(L)) = h(\bar{\theta}(R))$, de modo que $\bar{\theta}(L) \equiv_A \bar{\theta}(R)$ pelo Lema 5A.

□

Podemos, agora, apresentar a demonstração do teorema 5 , qual seja: existe um único homomorfismo g de $T_{\Sigma,\xi}$ em A, sendo A uma Σ-ξ-álgebra na classe $Alg_{\Sigma,\xi}$. Esquematicamente:



Prova do teorema 5:

Seja A uma Σ-álgebra qualquer satisfazendo ξ, e os homomorfismos $h_A : T_\Sigma \to A$ e $h_\xi = T_\Sigma \to T_{\Sigma,\xi}$ (pelo Lema 5A, h_ξ leva tem $[t]$). Agora, se existe um homomorfismo $g: T_{\Sigma,\xi} \to A$, a composição $g \circ h_\xi : T_\Sigma \to A$ é também um homomorfismo e, pela unicidade (teorema 2), $g \circ h_\xi = h_A$, isto é, $h_A(t) = g(h_\xi(t)) = g([t])$. Assim, temos que se g existe ele é único e $g([t]) = h_A(t)$. Assumindo ser esta a definição de g, vamos verificar que ela é independente da escolha do representante para classe da equivalência, isto é, se $[t] = [t']$, então $h_A(t) = h_A(t')$. Isto, porém, é precisamente o Lema 5C. Finalmente, a propriedade de homomorfismo para g decorre diretamente da definição de quociente e do fato de que h_A é um homomorfismo: $g(\sigma_{T/\equiv}([t_1],\ldots,[t_n])) = g([\sigma_T(t_1,\ldots,t_n)]) =$
$= \sigma_A(h_A(t_1),\ldots,h_A(t_n)) = \sigma_A(g([t_1]),\ldots,g([t_n]))$.

□

Com o teorema 5 , concluímos a apresentação dos resultados de Álgebra Universal que justificam o tratamento de tipos abstratos de dados como álgebras. No que se segue vamos definir o que seja uma especificação para um tipo abstrato e, visto ter um papel fundamental na metodologia de prova de correção da especificação, vamos também definir a álgebra dos termos canônicos, apresentando em seguida alguns resultados com relação a esta.

3. Especificação de Tipos Abstratos de Dados

Álgebras iniciais são geralmente objetos infinitos. Desse modo, se desejamos usá-las como tipos abstratos de dados, necessitamós de uma maneira conveniente para descrevê-las. Esse é o objetivo das especificações, para as quais apresentamos a seguinte definição:

> Definição 14. Uma especificação é uma tripla $\langle S, \Sigma, \xi \rangle$ onde Σ é um domínio de operadores S-sortido e ξ é um conjunto de Σ-equações.

Geralmente é nosso desejo especificar conjuntamente todos os sortes de S. Porém, a definição acima pode ser facilmente modificada para cobrir os casos em que se quer especificar só um ou um número $d_1, d_2, \ldots, d_n$ de sortes de S.

A idéia fundamental aqui é que $\langle S, \Sigma, \xi \rangle$ especifique um tipo abstrato de dados por definir $T_{\Sigma,\xi}$ como um representante para a classe de isomorfismo de Σ-álgebras.

Uma vez que é contra esta que deve ser provada a correção da implementação do tipo, a especificação precisa, antes de tudo, ser correta. Tal correção é verificada provando-se a existência de um isomorfismo entre $T_{\Sigma,\xi}$ e o modelo (uma Σ-ξ-álgebra) que se quer especificar.

4. Álgebra dos Termos Canônicos

O método devido a Goguen (Goguen-Thatcher-Wagner [10]) para demonstrar a correção de especificações algébricas se baseia na escolha de um representante canônico para cada classe de equivalência da álgebra $T_{\Sigma,\xi}$ e construção de uma nova Σ-álgebra C, chamada de álgebra dos termos canônicos, cujos domínios C_s são os conjuntos dos termos canônicos escolhidos. A demonstração se completa ao se mostrar que C e $T_{\Sigma,\xi}$ são isomorfas.

Em princípio, a escolha dos termos canônicos é totalmente arbitrária, mas na prática é necessário se fazer sua seleção de um modo disciplinado a fim de se tirar proveito da estrutura dos termos na prova de correção. A definição dada a seguir provê essa disciplina.

> Definição 15. Uma Σ-álgebra C é uma álgebra de Σ-termos canônicos se $C_s \subseteq (T_\Sigma)_s$, para cada $s \in S$, e se $\sigma(t_1,\ldots,t_n) \in C$ então $t_i \in C_{si}$ e $\sigma_C(t_1,\ldots,t_n) = \sigma(t_1,\ldots,t_n)$.

Um resultado importante é que dada uma especificação, temos, associado a esta, uma álgebra de termos canônicos, conforme assevera o teorema seguinte:

Teorema 6

Seja $<S,\Sigma,\xi>$ uma especificação. Associada a esta existe uma Σ-ξ-álgebra C de termos canônicos que é inicial em $Alg_{\Sigma,\xi}$.

Prova:

Suponha que podemos determinar um subconjunto $C \subseteq T_\Sigma$ tal que

a) Para cada classe de equivalência $[t] \in T_{\Sigma,\xi}$, C contém um único elemento $t^* \in [t]$, e

b) Para cada $\sigma \in \Sigma$, se $\sigma(t_1,\ldots,t_n) \in C$, então $t_i \in C$.

Por (a) nós podemos tornar C uma Σ-ξ-álgebra, definindo, para $t_i \in C$,

$$\sigma_C(t_1,\ldots,t_n) = (\sigma(t_1,\ldots,t_n))^*$$

Agora, para que C seja uma álgebra de termos canônicos, nós precisamos ter $\sigma_C(t_1,\ldots,t_n) = \sigma(t_1,\ldots,t_n)$ para $\sigma(t_1,\ldots,t_n) \in C$. Porém se $\sigma(t_1,\ldots,t_n) \in C$, (a) nos dá que $\sigma(t_1,\ldots t_n) = (\sigma(t_1,\ldots,t_n))^*$, o que é suficiente pois $\sigma_C(t_1,\ldots,t_n) = (\sigma(t_1,\ldots,t_n))^*$.

Com as hipóteses acima podemos mostrar que $T_{\Sigma,\xi}$ é isomorfo a C, onde o isomorfismo é a função $j : [t] \to t^*$. Que j é um-a-um e sobre é evidente. Devemos mostrar, no entanto, que j é um homomorfismo de $T_{\Sigma,\xi}$ em C. De fato, para $\sigma \in \Sigma_{s_1 \ldots s_n,s}$ e

$[t_i] \in (T_\Sigma, \xi)_{s_i}$, temos que $j(\sigma_{T/\equiv}([t_1],\ldots,[t_n])) =$
$= j([\sigma(t_1,\ldots,t_n)]) = (\sigma(t_1,\ldots,t_n))^* = (\sigma(t_1^*,\ldots,t_n^*))^* =$
$= \sigma_C(t_1^*,\ldots t_n^*) = \sigma_C(i_j[t_1],\ldots,j[t_n])$ (observe que $t_i \equiv_\xi t_i^*$).

Agora, para provar o teorema é suficiente exibir um conjunto C que satisfaça (a) e (b). Para isso vamos definir uma família $\langle C_n \mid n \in N \rangle$ de subconjuntos de T_Σ de modo que $C = U\{C_n \mid n \in N\}$ seja o conjunto de representantes canônicos desejado. Construiremos C_n por indução na profundidade dos termos, tal que

i) Se $t \in C_n$, então profundidade(t) $\leq n$;

ii) Se $t \in T_\Sigma$ é tal que $[t]$ tem um representante de profundidade $\leq n$, então existe um único representante para $[t]$ em C_n; e

iii) Se $\sigma(t_1,\ldots,t_n) \in C_n$, então $t_1,\ldots,t_n \in C_{n-1}$.

Seja C_0 qualquer subconjunto de $U_S \Sigma_{\lambda,s}$ tal que para cada $\sigma \in U_S \Sigma_{\lambda,s}$ existe um único $\sigma^* \in C_0$ com $\sigma^* \equiv_\xi \sigma$. Então os únicos termos de profundidade 0 são constantes e as condições (i), (ii) e (iii) são satisfeitas.

Suponha, agora, que C_n tenha sido definido satisfazendo (i), (ii) e (iii). Seja T_{n+1} o conjunto de classes de equivalência $[t]$ tal que existe $t^* \equiv_\xi t$ com profundidade(t*)=n+1, e para todo t', se $t' \equiv_\xi t$, então profundidade(t') $\geq$ n+1 (Existe um representante de profundidade n+1, e todos os representantes tem profundidade $\geq n+1$). Tomemos como único representante em C_{n+1} para o termo $\sigma(t_1^*,\ldots,t_n^*)$ de $[t] \in T_{n+1}$, o qual tem profundidade n+1 e para o

qual $t_i^* \in C_n$. Isso é possível uma vez que $t^* \in [t] \in T_{n+1}$ é de profundidade n+1 por definição, e $t^* = \sigma(t_1,\ldots,t_n)$. Porém profundidade$(t_i) \leq n$, e assim por (ii) $t_i \equiv_\xi t_i^* \in C_n$, de modo que $\sigma(t_i^*,\ldots,t_n^*)$ pode ser escolhido como representante de $[t]$.

Como conclusão temos que $C = U\langle C_n \mid n \in N\rangle$ contém um único representante t^* para cada $[t] \in T_{\Sigma,\xi}$ e, adicionalmente, que se $\sigma(t_1,\ldots,t_n) \in C$, então $t_1, t_2,\ldots,t_n \in C$.

□

Uma vez mostrado que existe uma Σ-ξ-álgebra que é inicial e ao mesmo tempo uma álgebra de termos canônicos, vamos demonstrar uma condição que torna $T_{\Sigma,\xi}$ e C (isomorfas) (Notação: $C \cong T_{\Sigma,\xi}$).

Teorema 7

Seja $\langle S,\Sigma,\xi\rangle$ uma especificação e C uma álgebra de Σ-termos canônicos. Então $C \cong T_{\Sigma,\xi}$ se e somente se

i) C satisfaz ξ

ii) Para cada $\sigma \in \Sigma_{s_1\ldots s_n,s}$ e $t_i \in C_{s_i}$,

$$\sigma(t_1,\ldots,t_n) \equiv_\xi \sigma_C(t_1,\ldots,t_n).$$

Antes de apresentarmos a demonstração desse teorema vamos anunciar e provar 2 lemas usados no mesmo.

Lema 7A

Seja $<S,\Sigma,\xi>$ uma especificação, A uma Σ-ξ-álgebra e $\equiv_A$ a congruência gerada em T_Σ pelo único homomorfismo $h_A: T_\Sigma \to A$. Se $\equiv_A \subseteq \equiv_\xi$ e h_A é sobrejetiva, então $A \cong T_{\Sigma,\xi}$.

Prova:

Uma vez que A satisfaz ξ, $\equiv_\xi \subseteq \equiv_A$ pelo lema 5C. Então pela hipótese do lema que queremos provar, temos que $\equiv_\xi = \equiv_A$, e assim $T_{\Sigma,\xi} = T_\Sigma/\equiv_\xi = T_\Sigma/\equiv_A$. A atribuição $[t] \in T_\Sigma/\equiv_A \to h_A(t) \in A$ é injetiva por definição de $\equiv_A$, sobrejetiva por hipótese, e um Σ-homomorfismo pelo lema 5A. Assim $T_{\Sigma,\xi} = T_\Sigma/\equiv_A \cong A$.

Lema 7B

Se C é uma Σ-álgebra de termos canônicos, então o único homomorfismo $h_C: T_\Sigma \to C$ é sobrejetivo; na realidade $h_C(t) = t$, para $t \in C$.

Prova:

Usando indução estrutural, se σ está em C e em $\Sigma_{\lambda,s}$ para algum $s \in S$, então $h_C(\sigma) = \sigma_C = \sigma$ pela definição 15. Indutivamente, suponha $h_C(t_i) = t_i$ para $t_i \in C$, $i = 1,2,\ldots,n$ e tome $\sigma(t_1,\ldots,t_n) \in C$. Assim temos que $h_C(\sigma(t_1,\ldots,t_n)) = \sigma_C(h_C(t_1),\ldots,h_C(t_n)) = \sigma_C(t_1,\ldots,t_n) = \sigma(t_1,\ldots,t_n)$, novamente usando a definição de

álgebra canônica.

Prova do Teorema 7:

Seja $h: T_\Sigma \to C$ o homomorfismo dado pela inicialidade de T_Σ. Para mostrar que $C \cong T_{\Sigma,\xi}$ nós precisamos, pelo lema 7A, somente mostrar que h é sobrejetivo e $\equiv_h \subseteq \equiv_\xi$. A primeira condição é justamente o asseverado pelo lema 7B.

Para mostrar que $\equiv_h \subseteq \equiv_\xi$ é suficiente provar que, para qualquer $t \in T_\Sigma$, $t \equiv_\xi h(t)$, porque se $t \equiv_h t'$, então $t \equiv_\xi h(t) = h(t') \equiv_\xi t'$ portanto $t \equiv_\xi t'$. Com isso nós temos que t é ξ-congruente com seu representante canônico.

Vamos, então, mostrar que $t \equiv_\xi h(t)$ por indução estrutural em t. Se $t = \sigma \in \Sigma_{\lambda,s}$, para algum $s \in S$, então $h(\sigma) = \sigma_C \equiv_\xi \sigma$ por hipótese. Agora seja $t = \sigma(t_1,\ldots,t_n)$, onde $\sigma \in \Sigma_{s_1\ldots s_n,s}$ e $t_1,\ldots,t_n \in T_\Sigma$ tal que $t_i \equiv_\xi h(t_i)$. Então $h(\sigma(t_1,\ldots,t_n)) = \sigma_C(h(t_1),\ldots h(t_n)) \equiv_\xi \sigma(h(t_1),\ldots,h(t_n)) \equiv_\xi \sigma(t_1,\ldots,t_n)$, pela hipótese do teorema e por indução.

Agora vamos provar o inverso, ou seja, se $C \cong T_{\Sigma,\xi}$, então (i) e (ii) ocorrem. De fato, por C ser inicial na categoria $Alg_{\Sigma,\xi}$, C é uma Σ-ξ-álgebra e assim satisfaz ξ que é o exigido por (i). Por outro lado, se $C \cong T_{\Sigma,\xi}$, temos que $\equiv_\xi = \equiv_C$, de modo que por (ii) nós precisamos mostrar que $\sigma(t_1,\ldots,t_n) \equiv_C \sigma_C(t_1,\ldots,t_n)$ para $t_i \in C$. Mas $h_C(\sigma(t_1,\ldots,t_n)) = \sigma_C(h(t_1),\ldots,h(t_n)) = \sigma_C(t_1,\ldots,t_n) = h_C(\sigma_C(t_1,\ldots,t_n))$, as duas últimas igualdades pe

lo lema 7B.

O teorema 7 se aplica diretamente na prova de correção da especificação, pois uma vez que se tenha uma álgebra de termos canônicos isomorfa a $T_{\Sigma,\xi}$, basta se provar que essa álgebra de termos canônicos é isomorfa ao modêlo que se pretende especificar Essa prova é, de um modo geral, simples.

Para maiores detalhes do exposto neste apêndice vê [10].

REFERÊNCIAS BIBLIOGRÁFICAS

[01] CARVALHO, R.L., Maibaum, T.S.E., Pequeno, T.H.C., Borquez, A.A.P. e Veloso, P.A.S., "A Model Theoretic Approach to the Theory of Abstract Data Types and Data Structures", Research Report CS-80-22, University of Waterloo, Ontario, Canadá, 1980.

[02] DAHL, O.J. e Hoare, C.A.R., "Hierarchical Program Structures", em STRUCTURED PROGRAMMING, Academic Press, London e New York, 1972, pp. 175-220.

[03] DENNIS, J., "An Example of Programming With Abstract Data Types", SIGPLAN Notices, Vol. 10, nº 7, 1975, pp. 25-29.

[04] EHRIG, H., Kreowski. H.-J. e Padawitz, P., "Some Remarks Concerning Correct Specification and Implementation of Abstract Data Types", Informatik Research Report nº 77-13, Technical University Berlin, 1977.

[05] EHRIG, H., Kreowski, H.-J. e Padawitz, P., "Stepwise Specification and Implementation of Abstract Data Types", Informatik, Technical University Berlin, 1978.

[06] FLON, L., "Program Design With Abstract Data Types", Department of Computer Science , Carnegie-Mellon University, Pittsburgh, Pa., 1975.

[07] FLON, L. e Misra, J., "A Unified Approach to the Specification and Verification of Abstract Data Types", IEEE Transactions on Software Engineering, Vol., nº 1979.

[08] FLON, L. "A Survey of Some Issues Concerning Abstract Data Types", Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., 1974

[09] GAUDEL, M.C., "Algebraic Specification of Abstract Data Types", IRIA Rapport de Recherche nº 360,Rocquencourt, 1979.

[10] GUGUEN, J.A., Thatcher, J.W. e Wagner, E.G., "An Initial Algebra Approach to the Specification, Correctness,and Implementation of Abstract Data Types", em CURRENT TRENDS IN PROGRAMMING METHODOLOGY, Vol. IV, Prentice-Hall, 1978, pp. 81-149.

[11] GUGUEN, J.A., "Abstract Erros for Abstract Data Types" , em FORMAL DESCRIPTION OF PROGRAMMING CONCEPTS, North Holland, Amsterdan, 1978, pp. 491-525.

[12] GUTTAG, J., "Abstract Data Types and the Development of Data Structures", CACM, Vol. 20, nº 6, 1977, pp. 396 - 404.

[13] GUTTAG, J., Horowitz, E. e Musser, D., "The Design of Data Type Specifications", em CURRENT TRENDS IN PROGRAMMING METHODOLOGY, Vol. IV, Prentice-Hall, 1978, pp. 60-79.

[14] GUTTAG, J., Horowitz, E. e Musser, D., "Abstract Data Types and Software Validation", Comm. ACM, Vol. 21, nº 12, 1978, pp. 1048-1064.

[15] GUTTAG, J. e Horning, J.J., "The Algebraic Specification of Abstract Data Types", Acta Informatica, Vol. 10, nº 1, 1978, pp. 27-52.

[16] HOARE, C.A.R., "Proof of Correctness of Data Representations" Acta Informatica 1, 1972, pp. 271-281.

[17] HOARE, C.A.R., "Proof of a Structured Program: The Sieve of Eratosthenes" The Computer Journal, Vol. 15, nº 4 , 19, pp. 321-325.

[18] HOARE, C.A.R., "Notes on Data Structuring", em STRUCTURED PROGRAMMING, Academic Press, London e New York, 1972 , pp.83-174

[19] INGARGIOLA, G.P., "Implementations of Abstract Data Ty - pes", California Institute of Technology Pasadene, California 91125, 1974.

[20] KAMIN, S., "Some Definitions for Algebraic Data Type Specifications", SIGPLAN, Vol. 14, nº 3, 1979, pp.28-37.

[21] LEDGARD, H., "ADA: AN INTRODUCTION", Springer-Verlag, New York, 1981.

[22] LISKOV, B., e Zilles, S., "Programming with Abstract Data Types", Proccedings of a Symposium on Very High Level Languages, SIGPLAN Notices, Vpl. 9, nº 4, 1974, pp. 50-59.

[23] LISKOV, B., Zilles, S., "An Approach to Abstraction" , Computation Structures Group Memo 88, MIT, Cambridge , Massachusetts, 1973.

[24] LISKOV, B., "Data Types and Program Correctness", SIGPLAN Notices, Vol. 10, nº 7, 1975, pp.16-17.

[25] LISKOV, B., Snyder, A., Atkinson, R., Schaffer, C. "Abs - traction Mechanisms in CLU, Comm. ACM, Vol. 20, nº 8, 1977, pp.564-576.

[26] LISKOV, B. e Zilles, S., "An Introduction to Formal Specifications of Data Abstractions", em CURRENT TRENDS IN PROGRAMMING METHODOLOGY, Vol. I, Prentice-Hall, 1978.

[27] LUCENA, C.J.P., e Pequeno, T.H.C., "Program Derivation Using Data Types: A Case Study", IEEE Transaction on Software Engineering, Vol. Se-5, nº 6, 1979, pp.586-592.

[28] LUCENA, C.J.P., ANALISE E SÍNTESE DE PROGRAMAS: UMA INTRODUÇÃO, Escola de Computação de Campinas, 1981.

[29] MANA, Z., MATHEMATICAL THEORY OF COMPUTATION, McGraw-Hill Book Company, New York, 1974.

[30] PEQUENO, T.H.C., "Uma Abordagem Lógica ao Problema de Representação de Tipos de Dados", Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 1978.

[31] PEQUENO, T.H.C. e Lucena, C.J., "An Approach for Data Type Specification and Its Use in Program Verification", Information Processing Letters, Vol. 8, nº2, 1979, pp. 98-103.

[32] PEQUENO, T.H.C., "Uma Descrição Formal dos Processos de Especificação e Implementação de Tipos Abstratos de Dados", Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Tese de Doutorado , 1981.

[33] PEREDA, A.A., "Métodos de Descrição de Tipos de Dados e Estruturas de Dados", Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Tese de Doutorado, 1979.

[34] PESSOA, F.E.P., e Veloso, P.A.S., "Introdução à Programação com Tipos Abstratos de Dados", Monografias em Ciência da Computação, nº12, Departamento de Informática , Pontifícia Universidade Católica do Rio de Janeiro , 1982.

[35] SHAW, M., Wulf, W. e London, R., "Abstraction and Verification in ALPHARD: Interation and Generators", Techi - cal Reports, Carnegie-Mellon University, Pittsburgh , Pa. e USC Information Sciences Institute, Marina Del Rey, California, 1976.

[36] VELOSO, P.A.S. e Pequeno, T.H.C., "Don't Write More Axioms Than You Have to: A Methodology for the Complete and Correct Specification of Abstract Data Types; With Examples", Monografias em Ciência da Computação, nº10, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

[37] VELOSO, P.A.S., "Namable Models and Programming", Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 1979.

[38] VELOSO, P.A.S. e Pequeno, T.H.C., "Interpretations Between Many-Sorted Theories", II Encontro Brasileiro de Lógica, Brasil, 1978.

[39] WIRTH, N., PROGRAMAÇÃO SISTEMÁTICA, Editora Campus Ltda., Tradução de Paulo A.S. Veloso, 1978.

[40] WULF, W., London, R. e Shaw,M., "Abstraction and Verification in ALPHARD: Introduction to Language and Methodology", Technical Reports, Carnegie-Mellon University, Pittsburgh, Pa. and USC Information Sciences Institute, Marina Del Rey, California, 1976.

[41] WULF, W., London, R. e Shaw, M., "An Introduction to the Construction and Verification of ALPHARD Programs" , IEEE Transactions on Software Engineering, Vol. Se-2 , nº 4, 1976, pp.253-265

[42] ZILLES, S., "Algebraic Specification of Data Types", Computation Structures Group Memo 119, MIT, 1975.