UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DEFINIÇÃO FORMAL DE TIPOS ABSTRATOS DE DADOS ATRAVÉS DE UM EXEMPLO

por

ELIZABETH SUELI SPECIALSKI

Dissertação submetida como requisito parcial para obtenção do grau de Mestre em Ciência da Computação

Laira Vieira Toscani

Profa. LAIRA VIEIRA TOSCANI

Orientadora

Porto Alegre, março de 1981

AGRADECIMENTOS

À professora Laira Vieira Toscani pela orientação segura e dedicada e pela confiança depositada.

Ao professor José Mauro Castilho pelos valiosos esclarecimentos.

À Alice e Jorge pela acolhida carinhosa e amiga.

À Márcia pela dedicação na datilografia.

Ao Celso

finalmente !

SUMARIO

SINOPSE

Este trabalho contém os conceitos básicos da área da abstração de dados e descrição de várias técnicas para especificação de tipos abstratos, sendo que um enfoque especial é dado para a técnica algébrica-axiomática. Um exemplo completo, a Grid, é apresentado desde a sua especificação até a prova da correção de sua implementação com o objetivo de tratar problemas decorrentes da abstração de tipos limitados e estáticos, bem como suas respectivas soluções, uma vez que estes problemas, devido à sua complexidade, não são comumente abordados.

ABSTRACT

This paper is concerned with the basic concepts of data abstraction and describes several techniques for specifying abstract data types; especially it focusses on the algebraic-axiomatic technique. A complete example; the Grid, is presented from its specification to the proof of correctness of its implementation. The main goal is to deal with problems stemming from abstracting limited and static data types. These problems, due to its complexity, are not commonly discussed in the available literature.

1 INTRODUÇÃO

Partindo-se de um conceito é possível encontrar vários programas que o implementem de forma correta. Na prática, em geral, os conceitos são estabelecidos informalmente. Se técnicas formais para demonstrar a correção da implementação de um conceito devem ser utilizadas, então é necessário inserir uma especificação entre o conceito e o programa que o implementa; a especificação consiste de uma descrição matemática do conceito e a correção de um programa é estabelecida provando que ele satisfaz a especificação.

O estudo de tipos de dados abstratos na Ciência da Computação teve sua importância aumentada, nos últimos anos, devido ao crescente interesse no uso da abstração como ferramenta para o desenvolvimento de grandes sistemas de software. A abstração permite reduzir a quantidade de detalhes que devem ser considerados num dado momento do projeto.

O uso de abstração facilita a verificação da correção de programas, clarifica o problema, melhora a comunicação entre projetistas e entre projetistas e implementadores.

Facilita a verificação da correção de programas, porque providencia uma especificação entre o conceito e o programa. E conforme Hoare [14], se o programador expressa o seu algoritmo como um programa abstrato operando sobre dados abstratos, adiando a decisão de uma representação concreta dos dados, a tarefa de provar a correção do programa concreto final fica facilitada (pois, se a representação dos dados é provada correta, basta provar a correção do programa abstrato original).

Torna mais claro o problema que deve ser programado, pois permite separar as propriedades essenciais das não essenciais num dado instante no desenvolvimento de um projeto. Além disso, dá uma visão clara descompromissada do problema, evitando uma escolha apressada e inadequada de implementação.

Proporciona um meio de comunicação entre os projetistas pois fornece uma especificação do conceito independentemente de sua representação. Pelo mesmo motivo facilita a comunicação entre projetistas e implementadores.

Estes fatos são importantes porque propiciam um aumento na confiabilidade do software.

Em muitas aplicações a complexidade dos dados contribui substancialmente na complexidade do programa que manipula os mesmos e esta é a razão pela qual a abstração de dados vem sendo introduzida em linguagens de programação.

Este trabalho trata de técnicas de especificação formal para tipos abstratos de dados, com ênfase para a técnica algébrica-axiomática definida por Guttag [10,11]. Esta técnica é ilustrada através de um exemplo completo para o qual é apresentado o conceito intuitivo, especificação formal, implementação e prova da correção da implementação.

Apresenta-se no capítulo 2, alguns termos técnicos e definições importantes extraídas da bibliografia consultada que são necessárias ao estudo de tipos abstratos de dados.

O capítulo 3 apresenta, suscintamente, algumas técnicas para especificação de tipos abstratos de dados conforme a divisão fornecida por Liskov e Zilles [21], com alguns exemplos de sua utilização. A técnica algébrica-axiomática definida por Guttag [11] é considerada com mais detalhes, uma vez que ela serve

de suporte para o desenvolvimento do exemplo. São considerados, também neste capítulo, aspectos referentes à consistência e completude da especificação algébrica-axiomática.

São exibidas, no capítulo 4, especificações de estruturas de dados conhecidas. São considerados os problemas decorrentes da especificação de tipos limitados sejam eles dinâmicos ou estáticos. Uma estrutura de dados mais complexa , a Grid, é definida e é apresentada a prova de que o conjunto dos axiomas de sua especificação é suficientemente-completo, de acordo com Guttag [10].

No capítulo 5, a Grid é implementada sobre outros tipos abstratos de dados cujas especificações e implementações também são fornecidas. A técnica utilizada para provar a correção da implementação de um tipo abstrato de dados é apresentada através de um exemplo de implementação de uma operação definida para a Grid.

A prova completa da correção da implementação da Grid é apresentada no Apêndice 1 e a prova da correção da implementação dos tipos abstratos definidos para auxiliarem a implementação da Grid é detalhada no Apêndice 2.

O Apêndice 3 é constituído por um conjunto de termos utilizados mas não definidos explicitamente no texto.

O Apêndice 4 contém um índice dos nomes das operações e funções utilizadas na especificação e implementação dos tipos abstratos Grid, Delgrid, Retanggr, List, Retanglist, Retang e Point, utilizados no desenvolvimento do exemplo.

2 DEFINIÇÕES

Neste capítulo serão apresentadas definições importantes relacionadas com a especificação de tipos abstratos.

Abstração: consiste em se considerar um conceito de forma independente de sua representação, isto é, é um mecanismo que permite separar as propriedades essenciais do conceito das propriedades associadas a sua representação, reduzindo os detalhes que devem ser considerados em um dado instante.

A principal conseqüência da abstração é a metodologia de desenvolvimento de projetos de sistemas que ela sugere.

Um problema que surge no desenvolvimento de projetos de grandes sistemas de software é a redução da complexidade ou da quantidade de detalhes que devem ser considerados de uma vez. Dois métodos de solução são a decomposição e a abstração. Através da decomposição uma tarefa fica dividida em duas ou mais subtarefas, porém, nem sempre esta decomposição é suficiente, pois, para muitos problemas, as subtarefas ainda ficam muito complexas. A complexidade dessas subtarefas pode ser reduzida através da abstração. Portanto, para reduzir a complexidade de um projeto top-down é necessário construir, a cada nível de refinamento, abstrações que suprimam todos os detalhes irrelevantes para aquele nível enquanto expõem claramente os conceitos e estruturas relevantes. Assim, pode-se reduzir o número de decisões que devem ser tomadas em um determinado instante.

Objeto de Dados: é um termo que se refere a uma classe de valores.

Estrutura de Dados: é um conjunto de elementos e uma coleção de relações estruturais entre esses elementos.

Tipos de Dados: Segundo Hilfinger [13] um tipo de dados é visto basicamente como a propriedade de um objeto que define o seu possível comportamento. Mais formalmente, um tipo caracteriza os valores possíveis de um objeto e o conjunto de operações que podem ser aplicadas a ele. Um tipo de dados pode, portanto, ser definido como um par formado por um objeto de dados e o conjunto de operações que tem significado para este objeto.

Tipo Abstrato de Dados: é um tipo de dados definido por uma especificação independente de sua representação, isto é, consiste na abstração de um tipo de dados. É também chamada de "abstração de dados".

Todas as técnicas de especificação para abstração de dados podem ser vistas como definindo uma disciplina matemática [21]. A disciplina surge a partir da especificação de um tipo abstrato de dados da mesma forma que a teoria numérica aparece a partir das especificações para os números naturais através dos axiomas de Peano.

O domínio da disciplina, isto é, o conjunto no qual ela é baseada, é a classe de valores pertencentes ao tipo abstrato de dados e as operações da abstração de dados são definidas como um mapeamento sobre este domínio.

A teoria da disciplina consiste dos teoremas e lemas deriváveis das especificações.

Todos os teoremas provados a partir de um conjunto de axiomas são verdadeiros para qualquer modelo que satisfaça os axiomas.

Uma técnica de especificação deve descrever a sintaxe e a semântica do tipo abstrato, isto é, deve apresentar as

operações, seus domínios, seus resultados, como os dados se relacionam e como são acessados. Esta descrição deve ser feita de tal forma que se permita ao implementador a liberdade de escolher a maneira como a semântica será realizada na máquina.

Liskov e Zilles [21] apresentam seis requisitos para que uma técnica de especificação possa ser útil. Estes critérios apresentam considerações práticas e teóricas já que o objetivo de uma técnica de especificação é permitir a escrita de especificações para programas práticos. De acordo com estes critérios, um método de especificação deve:

- ser formal;
- apresentar uma construção sem dificuldade excessiva;
- ter uma forma de apresentação que seja facilmente compreendida por pessoas que tenham conhecimento da notação usada;
- descrever apenas as propriedades que interessam ao conceito;
- ter um número relativamente grande de aplicações;
- permitir que uma pequena troca no conceito também resulte em uma troca pequena na sua especificação.

Implementação de uma Abstração de Dados: consiste de uma atribuição de significado aos valores e operações do tipo abstrato de dados em termos de valores e operações de um outro tipo abstrato ou conjunto de tipos abstratos, ou seja, considerando-se a existência de uma hierarquia de abstrações, onde as abstrações de um nível mais baixo fornecem explanações detalhadas para as operações que aparecem em uma abstração de um nível mais alto, pode-se dizer que as abstrações de nível mais baixo implementam a abstração de nível mais alto.

Outro aspecto a considerar na formalização é o de-

senvolvimento de metodologias para provar a correção do uso e da implementação de uma abstração de dados, dada a sua especificação.

Verificação: é o processo através do qual demonstra-se que a implementação de um tipo abstrato de dados satisfaz a sua especificação.

3 TÉCNICAS PARA ESPECIFICAÇÃO DE TIPO ABSTRATOS DE DADOS

O uso da abstração foi introduzido para resolver o problema criado pela crescente complexidade contida na maioria do software que se pode criar, que, por sua vez, é muito maior que a complexidade que uma pessoa pode captar em um determinado instante.

Metodologias de programação, voltadas para estrutura de algorítmos, como programação estruturada e programação top-down, foram desenvolvidas. Entretanto, um algorítmo é apenas um meio para se atingir o objetivo que é a transformação dos dados. A complexidade dos dados a serem manipulados aumenta substancialmente a complexidade do problema. Então mais recentemente estão sendo desenvolvidas a teoria dos tipos abstratos e metodologias de implementação de abstração de dados. Neste capítulo serão apresentadas algumas técnicas de especificação de tipos abstratos.

3.1 CLASSIFICAÇÃO DE MAJSTER

Segundo Majster [23], as técnicas existentes para descrever abstrações de tipos de dados podem ser agrupadas em duas classes:

a) - não construtivas: que caracterizam um tipo por certas propriedades e que podem ser classificadas em axiomáticas e algébricas. Na técnica axiomática, a implementação de um tipo pode ser vista como um modelo. Na técnica algébrica o tipo de dados

é definido somente através de suas operações que são mapeamentos satisfazendo certas relações. A técnica de especificação algébrica pode ser considerada um caso especial do método axiomático. Para uma especificação algébrica, os axiomas são apresentados em uma forma restrita, por exemplo: só são permitidas aquelas funções e predicados que representam operação explicitas para tipos de dados, o quantificador existencial não pode ocorrer e os axiomas devem ter a forma de equações com os lados esquerdo e direito de acordo com regras específicas.

b) - métodos construtivos que estabelecem explicitamente como os objetos se parecem e como as operações afetam estes objetos. Ex: métodos usando diferentes tipos de grafos ou automatas.

Outra classificação para técnicas de especificação é dada por Liskov e Zilles que as dividem em cinco categorias.

Apresenta-se, a seguir, cada uma das técnicas e as fontes onde poderão ser encontradas para um estudo mais minucioso.

3.2 Classificação de Liskov e Zilles

3.2.1 Disciplina Fixa

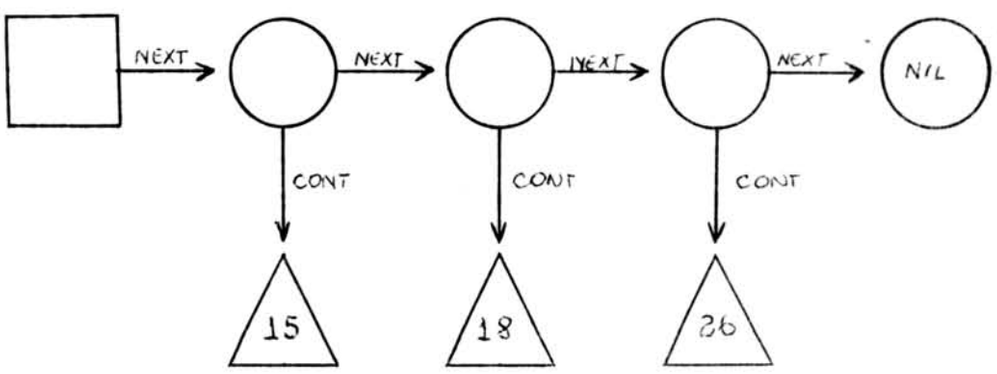
Esta técnica refere-se ao uso de um domínio fixo de objetos formais tais como conjuntos e grafos.

Earley [2] apresenta o uso de grafos na representação da semântica de estruturas de dados. Para ele, cada estrutura de dados é representada por um "V-graph" que é um grafo dirigi-

do com arcos rotulados. Cada nodo do grafo representa uma parte da estrutura de dados. Os arcos entre os nodos representam caminhos de acesso na estrutura e o arco de um nodo para um "átomo" significa que o átomo está armazenado neste nodo.

Como se pode observar, um V-graph possui três tipos de componentes: nodos, arcos e átomos. Átomos são objetos que não se dividem, isto é, não possuem partes que possam ser examinadas ou trocadas. As operações e testes permitidos sobre um átomo são aqueles que o consideram como um elemento indivisível (por ex. = e +). Os arcos são interpretados como seletores, isto é, cada rótulo de arco seleciona um único caminho de acesso a partir do nodo, e podem ser dirigidos de um nodo para outro ou de um nodo para um átomo. Os nodos não possuem um significado intrínseco; todo seu significado é derivado das relações estruturais e de acesso representadas pelos arcos. Cada nodo pode ter um número arbitrário de arcos originando-se nele, mas dois arcos que saem de um mesmo nodo não podem ter o mesmo nome seletor.

Uma lista encadeada contendo os átomos 15, 18 e 26 pode ser representada por



onde NEXT e CONT são rótulos que selecionam, respectivamente, o próximo nodo na lista e o conteúdo deste nodo; o nodo cabeça é o representado por um quadrado, os nodos intermediários por círculos e o nodo que representa o fim da lista por um círculo contendo a palavra reservada "NIL". Neste exemplo, as palavras NEXT e CONT não são necessariamente palavras reservadas pois elas poderiam ser substituídas por quaisquer outras duas seqüências de caracteres. Os triângulos representam os átomos da estrutura.

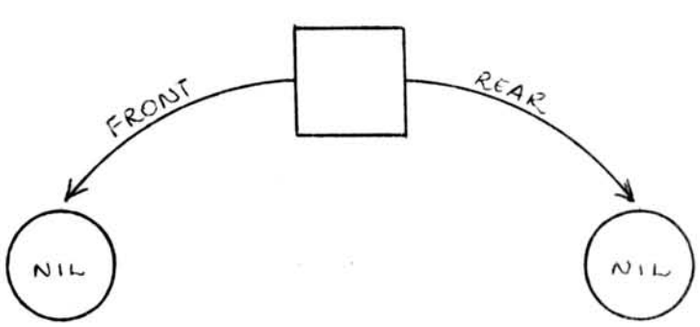
A manipulação de grafos exige que se defina um conjunto de operações primitivas, que devem ser escolhidas de acordo com o dado abstrato que se quer representar.

Como ilustração desta técnica apresenta-se a descrição de uma fila na qual um elemento é inserido no final e retirado na frente da fila.

As operações primitivas usadas neste exemplo são INITIAL, TRANS, INSERT e DELETE.

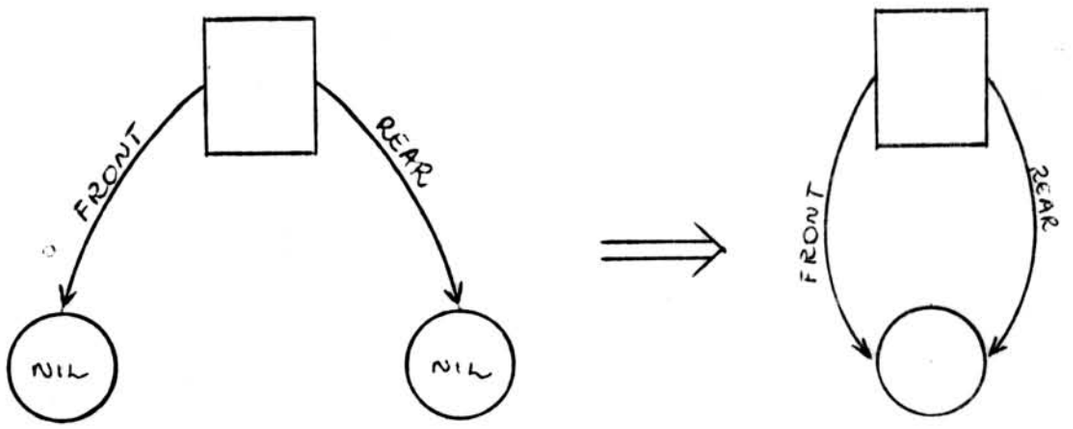
INITIAL - cria uma fila com um nodo cabeça onde se originam duas linhas seletoras FRONT e REAR. Estas linhas apontam para nodos com rótulos NIL indicando, desta forma, que não existe elemento na frente da fila nem tampouco no final.

INITIAL -



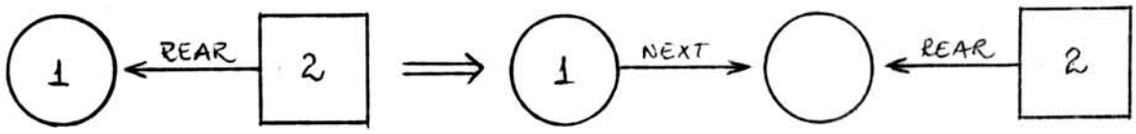
TRANS - insere o primeiro elemento na fila, isto é, o elemento inserido estará, ao mesmo tempo, na frente e na ré da fila.

TRANS



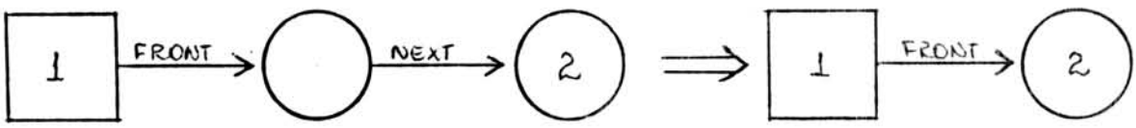
INSERT - insere um elemento no final da fila:

INSERT -



DELETE - retira o elemento que ocupa a posição apontada por FRONT

DELETE



Pode-se observar que esta técnica apresenta a desvantagem de que as especificações tornam-se muito longas e algumas apresentam muitas dificuldades para a definição. Liskov e Zilles [21] apresentam um exemplo onde estas desvantagens aparecem.

3.2.2 Disciplina Arbitrária

As técnicas que usam uma disciplina fixa para expressar especificações de tipos de dados podem ser comparadas com o uso de uma linguagem de programação que forneça um único método de estruturação de dados (de fato, Earley [2] define uma linguagem de programação, VERS, na qual V-graphs é um método de estruturação de dados). Embora o único método possa ser tão poderoso que permita a implementação de todas as estruturas de dados definidas pelo usuário, isto não implica que todas as estruturas sejam implementadas com igual facilidade. Da mesma forma, não se pode esperar que todas as abstrações de dados sejam igualmente bem especificadas em termos de um disciplina fixa.

Por outro lado, o uso de uma disciplina arbitrária pode ser comparado com o uso de uma linguagem de programação que forneça várias facilidades para estruturação de dados, pois, neste caso, permite-se que as especificações sejam escritas em qualquer disciplina conveniente. A experiência em programação mostra, no entanto, que sempre existirão abstrações que não podem ser representadas idealmente por qualquer dos métodos de estruturação de dados.

A técnica de uso de uma disciplina arbitrária é particularmente útil quando a classe dos objetos da abstração de dados desejada é um subconjunto de algum domínio matemático estabelecido.

Como ilustração desta técnica, considere a especificação do conjunto de inteiros através da teoria dos conjuntos, que é o domínio natural para operações em conjuntos

$$\text{EMPTY} \equiv \{\ \}$$

$$\text{INSERT}\ (s, i) \equiv s \cup \{i\}$$

$$\text{DELETE}\ (s, i) \equiv s \cap \neg \{i\}$$

$$\text{HAS}\ (s, i) \equiv i \in s$$

Para que a especificação acima defina conjuntos de tamanho limitado, basta que se adicione pré-condições para especificar quando é permitido aplicar uma operação. Um exemplo de pré-condição seria estabelecer que a operação INSERT só pode ser aplicada quando o tamanho do conjunto é menor que um determinado inteiro positivo N.

Quando pré-condições aparecem, é necessário mostrar que o uso de uma operação satisfaz as pré-condições ou resulta em um erro.

3.2.3 Modelo de Máquina de Estado

Nesta técnica, a máquina de estado é identificada como um único objeto representativo e a especificação descreve como o estado da máquina é modificado de acordo com o resultado

da aplicação de algumas operações.

As operações são classificadas em dois grupos: as V-operações, que não ocasionam troca de estado da máquina mas que permitem a observação de alguns aspectos do estado e as O-operações, que causam uma troca de estado.

A especificação de cada operação tem três partes: possíveis valores, parâmetros e efeito. Os possíveis valores tem significado apenas para as V-operações e determinam o tipo de valor retornado pela operação e sua inicialização quando a máquina é criada. Parâmetros especificam os argumentos de entrada da operação. O objeto do tipo a ser definido (no exemplo a seguir, o conjunto de inteiros) não é listado como um parâmetro porque a máquina é objeto. A parte "efeito" especifica o efeito da operação no estado da máquina de estados e as ações efetuadas quando erros são detectados. O efeito de uma O-operação é especificado em termos das V-operações.

Considere o conjunto de inteiros, definido anteriormente, na técnica da disciplina arbitrária. A especificação para este conjunto através do uso de uma máquina de estado é dada a seguir.

Especificação para o conjunto de inteiros através de uma máquina de estados:

V-operação: HAS

possíveis valores: boolean; inicialmente FALSE

parâmetros: Integer i

efeito: nenhum

O-operação: INSERT

possíveis valores: nenhum

parâmetros: Integer i

efeito: (∀j) HAS (j) = if j = i then TRUE

else 'HAS' (j)

O-operação: DELETE

possíveis valores: nenhum

parâmetros: Integer i

efeito: (∀j) HAS (j) = if j = i then FALSE

else 'HAS' (j)

O efeito de uma O-operação é especificado em termos das V-operações. O uso de 'HAS' entre aspas, na especificação de INSERT, significa o valor antigo de HAS (antes da execução de INSERT).

O efeito da operação EMPTY é dado implicitamente pela especificação do valor inicial da V-operação (no exemplo HAS tem valor inicial FALSE) já que ela significa a atribuição de um estado inicial à máquina de estado.

Nem sempre é fácil encontrar uma descrição simples de uma O-operação, já que algumas delas ocasionam efeitos retardados sobre as V-operações, isto é, algumas propriedades do estado serão verificadas pelas V-operações somente após a aplicação de uma O-operação. Um exemplo desta situação é dado por Liskov e Zilles [21] na abstração de dados "pilha" que é apresentada a seguir.

Especificação para a pilha através de uma máquina de estados:

V-operação: TOP

possíveis valores: Integer; inicialmente UNDEFINED

parâmetros: nenhum

efeito: chamada 1 de erro se 'DEPTH' = 0

V-operação: DEPTH

possíveis valores: Integer, inicialmente 0

parâmetros: nenhum

efeito: nenhum

O-operação: PUSH

possíveis valores: nenhum

parâmetros: Integer a

efeito: TOP = a

DEPTH = DEPTH + 1

O-operação: POP

possíveis valores: nenhum

parâmetros: nenhum

efeito: chamada 2 de erro se 'DEPTH' = 0

DEPTH = DEPTH - 1

a seqüência PUSH (a); POP não apresenta efeito se nenhuma chamada de erro ocorre.

Observe que a operação PUSH (que coloca um elemento no topo da pilha) tem efeito retardado sobre a operação TOP (que fornece o elemento do topo da pilha) pois o elemento situado na posição anterior ao topo da pilha não é diretamente observável visando-se TOP, mas será observável após a aplicação de POP (que retira o topo da pilha).

Outra observação importante a respeito da especificação dada para a pilha é que ela estabelece que rotinas para tratamento de erros serão chamadas caso elas ocorram. Por exemplo, a V-operação DEPTH foi inserida na especificação para controlar o número de elementos da pilha. A O-operação POP tem como efeito a chamada de uma rotina de erro caso ocorra a condição 'DEPTH' = 0 (Lembre que 'DEPTH' significa o valor anterior de DEPTH, isto é, antes de POP ser executado).

Ainda com respeito a especificação da pilha, observe que na parte "efeito" da operação POP, aparece a frase "a seqüência PUSH(a); POP não apresenta efeito se nenhuma chamada de erro ocorre" que expressa a tentativa de descrever o efeito retardado da operação PUSH sobre TOP.

Esta especificação pode ser modificada acrescentando-se as "propriedades de módulo", introduzidas por Parnas [25], que são aplicadas sobre todo o grupo de operações. Então, a parte efeito da operação POP conterá apenas a chamada de erro se 'DEPTH' = 0 e a decrementação DEPTH = 'DEPTH' - 1; e na especificação da pilha, logo após a definição da V-operação POP, será incluído:

Propriedades de módulo:
a seqüência PUSH(a), POP não apresenta efeito se nenhuma chamada de erro ocorre.

A técnica de especificação através de máquina de estado apresenta muitas desvantagens; entre elas destaca-se o fato de que a inclusão de uma nova V-operação pode requisitar atualizações em uma grande parte das especificações de O-operações, uma vez que o efeito das O-operações é dado em termos das V-operações.

3.2.4 Sistemas Axiomáticos

Nesta técnica a formalização para a abstração de dados é conseguida expressando-se as propriedades como axiomas.

Considere, como exemplo, a especificação axiomática para a abstração de Stack:

1. STACK (CREATE)
2. STACK (s) & INTEGER (i) → STACK (PUSH (s, i)) &
   [POP(s) ≠ STACKERROR → STACK (POP(s))]
   & [TOP(s) = INTEGERERROR → INTEGER (TOP (s))]
3. (∀P) P(CREATE) & (∀s) (∀i) [STACK (s) & INTEGER (i) & [P(s) → PUSH(s,i)] & [s ≠ CREATE → P(POP (s))]]
   → (∀s) [STACK (s) → P(s)]
4. STACK (s) & INTEGER (i) → TOP (PUSH(s,i)) = i
5. TOP (CREATE) = INTEGERERROR
6. STACK (s) & INTEGER (i) → POP(PUSH(s,i)) = s
7. POP (CREATE) = STACKERROR

As especificações axiomáticas podem, quase sempre, ser pouco extensas e amplamente aplicáveis, em parte porque existem poucas limitações na forma dos axiomas. A técnica também suporta a extensibilidade, isto é, para estender o conceito definido basta adiciomar novos axiomas ou, no máximo, modificar alguns dos axiomas existentes.

Para provar que uma implementação de uma abstração de dados, especificada por axiomas, é correta, deve-se mostrar que a implementação é um modelo dos axiomas.

Uma desvantagem desta técnica é que ela não define

diretamente um modelo para a classe de objetos, uma vez que é definida apenas implicitamente. Isto causa com que muitas vezes seja dificil ver que os axiomas realmente definem o conjunto de valores de interesse.

Guttag [9] apresenta a técnica de especificação axiomática de HOARE usando uma notação similar aos módulos de Euclid encontrada em Guttag e Horning [10].

3.2.5 Sistemas Algébricos

Nesta técnica, o conjunto de expressões legais é constituído por expressões que podem ser formadas, a partir das operações dadas para a abstração dos dados, de tal forma que a correção do tipo é preservada. Uma expressão "preserva" a correção do tipo se todo operador tem em seus operandos uma expressão cujo resultado combina com o tipo requerido para aquele operando.

As especificações algébricas apresentam uma analogia com as especificações da máquina de estado no sentido de que os efeitos das operações são dados como propriedades de módulos.

Liskov e Zilles [21] apresentam uma especificação algébrica para Stacks e chamam a atenção para o fato de que muitas vezes é necessária a inclusão de expressões condicionais na especificação. Isto indica que só são fornecidos resultados para as substituições que satisfaçam alguma condição.

Como exemplo, considere a especificação algébrica para a abstração do tipo Queue de inteiros.

funcionalidade:

NEWQ	→ Queue
ADDQ: Queue x Integer	→ Queue
DELETEQ: Queue	→ Queue
FRONTQ: Queue	→ Integer U{INTEGERERROR}
ISNEWQ: Queue	→ Boolean
APPENDQ: Queue x Queue	→ Queue

equações

1. ISNEWQ (NEWQ) = TRUE
2. ISNEWQ (ADDQ (q,i)) = FALSE
3. DELETEQ (NEWQ) = NEWQ
4. DELETEQ (ADDQ (q,i)) = IF ISNEWQ (q) THEN NEWQ
   ELSE ADDQ (DELETEQ (q), i)
5. FRONTQ (NEWQ) = INTEGERERROR
6. FRONTQ (ADDQ (q,i)) = IF ISNEWQ (q) THEN i
   ELSE FRONT (q)
7. APPENDQ (q, NEWQ) = q
8. APPENDQ (r, ADDQ (q,i)) = ADDQ (APPENDQ (r,q),i)

3.3 Técnica Algébrica - Axiomática

Um tipo abstrato é um conjunto de valores e operações definidas independentemente da sua representação.

Um sistema algébrico é também um conjunto de valores e operações.

Axiomas expressam abstratamente o comportamento e relações que caracterizam operações e classes de valores. Portanto, é natural representar uma abstração de dados, como um

sistema algébrico cujas operações são definidas por axiomas.

Majster [22] considera que a técnica algébrica é um caso particular da técnica axiomática onde os axiomas são apresentados obedecendo certas regras. Já Guttag [11], chama esta combinação de técnica "algébrica - axiomática".

A especificação de um tipo de dados através da técnica algébrica-axiomática compreende uma especificação sintática e uma especificação semântica.

Na especificação sintática são definidos os nomes, os domínios e os contradomínios das operações.

A especificação semântica contém o conjunto de axiomas, na forma de equações, caracterizando o inter-relacionamento entre as operações.

Um exemplo simples de especificação de um tipo abstrato através da técnica algébrica axiomática é a definição do tipo Stack, dada por Guttag [11].

```
type Stack [elementtype: Type]

sintaxe

        NEWSTACK   →    Stack
        PUSH  (Stack, elementtype)  →  Stack
        POP   (Stack)   →  Stack
        TOP   (Stack)   →  elementtype   U {UNDEFINED}
        ISNEW (Stack)   →  Boolean
        REPLACE (Stack, elementtype) →  Stack
```

semântica

declare stk: Stack, um: elementtype:

POP (NEWSTACK) = NEWSTACK

POP (PUSH (stk, um)) = stk

TOP (NEWSTACK) = UNDEFINED

TOP (PUSH (stk, um)) = um

ISNEW (NEWSTACK) = TRUE

ISNEW (PUSH (stk, um)) = FALSE

REPLACE (stk, um) = PUSH (POP (stk, um))

Com o objetivo de simplificar a especificação semântica, trabalhos mais recentes de Guttag acrescentam um conjunto de restrições que compreendem uma especificação de pré-condição e uma especificação de falha. As pré-condições limitam a aplicação dos axiomas enquanto que as falhas restringem o domínio de uma operação e servem para chamar a atenção do implementador sobre determinados aspectos a serem considerados.

3.3.1 Consistência e "Completeness"

Uma vez construída a especificação de um tipo abstrato, é importante verificar a consistência do conjunto de axiomas.

A semântica de um tipo abstrato especificado segundo a técnica algébrica-axiomática é descrita por um conjunto de axiomas. Se estes axiomas podem ser usados para formar uma sentença que contradiz um dos axiomas da especificação, o conjunto dos axiomas da especificação é inconsistente.

No caso geral, a determinação da consistência de um conjunto arbitrário de equações é um problema insolúvel (indecidível)[10]. Na prática, no entanto, é relativamente simples demonstrar a consistência de um conjunto particular. A construção de um modelo é talvez a técnica mais comumente usada. Para mostrar que a axiomatização de um tipo abstrato é consistente, é suficiente construir uma implementação da abstração e provar que esta implementação está correta. Do ponto de vista prático, esta é a melhor maneira de se demonstrar consistência. A desvantagem é que se a especificação é inconsistente, é possível dispender um esforço considerável na busca de um modelo que não pode ser encontrado.

Outro aspecto importante a ser considerado sobre o conjunto de axiomas é a "completeness". A definição usual de "completeness" em lógica é: "um conjunto de axiomas de um sistema é completo se e somente se toda fórmula bem formada ou sua negação pode ser provada como teorema do sistema". Na teoria dos tipos abstratos Guttag [9, 10] usa uma definição mais fraca para esta condição, que ele chama de "suficientemente-completa".

Antes de se definir o termo "suficientemente-completa", é necessário definir para um tipo abstrato T, o conjunto de operações O e o conjunto de palavras L(T):

- O é o conjunto das operações que executam o mapeamento do tipo T em outro tipo.
- L(T) é o conjunto de expressões que podem ocorrer em aplicações sucessivas de operações do tipo abstrato T.

Então, dado um tipo abstrato T e um conjunto de axiomas A, A é dito uma axiomatização de T suficientemente-com-

ta se e somente se para todo o $(e_1, e_2, \ldots, e_n) \in L(T)$, onde o $\in$ O, existe um teorema derivável de A da forma o(e_1,e_2,.., en) = u, onde u não contém operações do tipo T.

Informalmente pode-se afirmar que se a combinação dos axiomas não apresenta todas as informações necessárias para a captura do significado das operações do tipo abstrato, então a axiomatização não é suficientemente-completa.

De um modo genérico, o problema de se estabelecer se um conjunto de axiomas é suficientemente-completo ou não, é indecidível, porém Guttag [10] mostra que, limitando-se a maneira de se especificar o tipo abstrato, existem condições suficientes que garantem que a axiomatização é suficientemente-completa.

3.3.2 Correção da Implementação

A implementação de um tipo abstrato T consiste de duas partes: uma parte de programas e uma de representação.
- a parte de programas consiste em qualquer interpretação das operações de um tipo que resulte em um modelo para os axiomas da especificação de T;
- a representação é uma função que mapeia termos do domínio do modelo em seus representantes no domínio abstrato.

Uma implementação é correta se preserva os axiomas, isto é, as propriedades das operações.

A especificação de um tipo abstrato T pode ser vista como a descrição de uma algébra. Neste caso, uma implementação de T, feitas sobre um tipo abstrato T', é correta se é um

homomorfismo da algébra de T na algébra de T'.

3.3.3 A Linguagem de Especificação

A escolha da linguagem para expressar as especificações é importante. Ela deve proporcionar facilidades para provas de correção e portanto deve ser, por si só, definida axiomaticamente.

Neste trabalho assume-se uma linguagem base com cinco primitivas: composição de funções, relação de igualdade (=), duas constantes distintas (TRUE e FALSE) e um conjunto ilimitado de variáveis livres.

A partir destas primitivas pode-se construir uma linguagem de especificação na qual uma operação que tenha sido definida em termos das primitivas deve ser adicionada à linguagem de especificação.

A operação IF-THEN-ELSE, por exemplo, pode ser definida pelos axiomas:

IF-THEN-ELSE (TRUE, q, r) = q

IF-THEN-ELSE (FALSE, q, r) = r

Assume-se, então, que a operação IF-THEN-ELSE, escrita da forma IF b THEN q ELSE r, é parte da linguagem de especificação.

Assume-se, também, a disponibilidade dos operadores booleanos infixados $\wedge$, $\vee$, $\rightarrow$ (ou $\supset$), = (ou $\equiv$) e o operador prefixado $\neg$.

Permite-se, ainda, as operações convencionais sobre

os inteiros: adição (+), subtração (-), multiplicação (*), divisão (DIV) e resto da divisão (MOD).

3.3.4 Construção da Especificação de um Tipo Abstrato

Uma vez definida a linguagem que será utilizada para a especificação do tipo abstrato, resta estabelecer as convenções e regras a serem obedecidas.

A seleção dos axiomas apropriados depende, principalmente, do tipo a ser definido, Não existe um procedimento genérico para a construção de axiomatização para álgebras de tipo. É possível, no entanto, caracterizar a classe de informações que devem ser fornecidas por uma axiomatização, isto é, generalizar a forma que uma axiomatização deve tomar.

Qualquer processo que desejar fazer uso do tipo de dados definido deve poder fazer isto examinando apenas a especificação, isto é, não deve ser necessário compreender a implementação.

Uma boa especificação deve apresentar uma quantidade adequada de informações para permitir a definição do tipo, mas estas informações não devem restringir a forma de implementação.

Do ponto de vista computacional, é adequado definir um tipo de dados de forma construtiva, isto é, apresentar operações que criam, modificam e destroem configurações do tipo de dados.

Como exemplo de especificação de um tipo abstrato apresenta-se a definição de um array não limitado. A notação con

vencional exibida nesta especificação será usada em todos os exemplos posteriores [11]. Os nomes das operações serão escritos com todas as letras maiúsculas. O nome de um tipo de dados inicia com letra maiúscula. Nas equações as variáveis são escritas com letras minúsculas e devem ser declaradas de forma que se identifique a que tipo pertencem.

Para a especificação de um tipo abstrato, através da técnica-algébrica-axiomática, convenciona-se que existam duas etapas: a sintaxe e a semântica. A sintaxe é precedida por uma declaração.

- a declaração fornece o nome do tipo que se deseja definir e a classe de seus elementos. Por exemplo, na declaração:

type Array (domaintype: Integer, rangetype: Real)

especifica-se que o nome do tipo é Array e que seus elementos são bem definidos através de um valor inteiro (índice do array) e um valor real (conteúdo do array). As expressões domaintype e rangetype são reservadas para indicarem, respectivamente, o domínio e o contradomínio do tipo definido. Sendo assim, a declaração dada como exemplo, estabelece que o domínio do tipo Array é formado por inteiros e o contradomínio é formado por reais.

Caso se queira definir um Array de forma mais genérico, isto é, permitindo que os elementos do domínio e do contradomínio pertençam a qualquer tipo definido externamente, utiliza-se a notação convencional Type. Por exemplo, a declaração:

type Array (domaintype: Type, rangetype: Type)

permite que os elementos do domínio e do contradomínio sejam de qualquer tipo, mas todos do mesmo tipo para domaintype e todos do mesmo tipo para rangetype.

Se o array a ser especificado é bidimensional, seus

elementos devem ser identificados através de pares ordenados (i,j) e isto implica que a declaração do tipo seja dada por type Array [domaintype: Type x Type, rangetype: Type]

O uso da expressão Type x Type é convencional e significa que o domínio do tipo abstrato é composto por pares ordenados onde os dois elementos são do mesmo tipo.

Seguindo a declaração do tipo, descrita acima, aparece a parte sintática da especificação.

Na parte sintática são definidos os nomes das operações permitidas sobre o tipo, seus domínios e contradomínios.

Por exemplo, a sintaxe do tipo Array pode ser composta das operações:

NEWARRAY → Array

ASSIGN (Array, domaintype, rangetype) → Array

ACCESS (Array, domaintype) → rangetype U{UNDEFINED}

A operação NEWARRAY não possui operandos, isto é, ela serve para criar um Array enquanto que a operação ASSIGN tem como operandos um array, um elemento de domaintype e um elemento de rangetype e como resultado um novo array.

Já a operação ACCESS possui como operandos um ARRAY e um elemento de domaintype e devolve um elemento de rangetype ou uma mensagem de indefinido.

Observe também que, além de criar o Array, apenas mais duas operações são permitidas sobre o tipo Array: atribuir um valor a um elemento do Array (ASSIGN) e recuperar o valor de um elemento caso algum já lhe tenha sido atribuído (ACCESS).

Note que não existe preocupação com o tamanho do Array e portanto pode-se deduzir que trata-se da especificação de um tipo não limitado.

Como já foi dito anteriormente, a especificação semântica contém um conjunto de axiomas, na forma de equações, que relacionam as operações entre si.

Um Array é definido através de seus elementos e portanto a parte de especificação semântica deverá ser apresentada de tal forma que este aspecto transpareça. Desta forma, a semântica das operações será construída tendo por base a operação ACCESS. De acordo com o que foi apresentado na secção 3.3.1, esta é realmente a forma de se construir a parte semântica de uma especificação para que se garanta a condição de um conjunto de axiomas "suficientemente-completo", uma vez que a operação ACCESS tem um resultado diferente do tipo que está sendo definido.

Além disso, deve-se ter o cuidado de que as construções recursivas contidas nos axiomas sejam finitas. No caso do Array, os axiomas contém uma construção recursiva tendo como base de recursão o axioma ACCESS(NEWARRAY, d_1) = UNDEFINED. Assim é possível provar que ACCESS (arr,d_1) pode ser expresso sem conter operações definidas na especificação do Array.

Uma prova de que um conjunto de axiomas é suficientemente completo é apresentado na secção 4.6.

A parte semântica da especificação do Array é dada por:

declare arr: Array; d_1, d_2: domaintype

r: rangetype

ACCESS (NEWARRAY, d_1) = UNDEFINED

ACCESS (ASSIGN (arr, d_1, r), d_2) =

= IF d_1 = d_2

THEN r

ELSE ACCESS (arr, d_2)

Olhando a especificação sintática pode-se verificar que cada uma das expressões nas equações axiomáticas está formada corretamente no sentido de que cada operador está aplicado a um número correto de argumentos e cada argumento é do tipo correto.

4 O PROJETO DE ESPECIFICAÇÕES DE TIPOS ABSTRATOS

A especificação de um tipo abstrato dá uma definição formal de cada operação do tipo, de forma independente da implementação. O projeto completo de um tipo abstrato particular compreende, portanto, uma especificação seguida de uma implementação consistente com a especificação.

4.1 Considerações Gerais

Na aplicação de uma técnica de especificação é usual pressupor a existência de outros tipos abstratos definidos independentemente e usá-los na especificação. Segundo Guttag e Horning [10], teoricamente é possível definir vários tipos abstratos através de uma recursão mútua, embora muitas vezes uma separação dos tipos forneça uma especificação mais clara.

A maioria dos exemplos de especificação de dados abstratos, encontrados na literatura existente sobre o assunto, trata de tipos de dados não limitados. A justificativa para este fato é que os tipos limitados apresentam maior dificuldade de definição.

Um dos mais simples exemplos de um tipo abstrato de dados é a pilha. Guttag [11] apresenta uma especificação da pilha não limitada através de seis operações na especificação sintática e um conjunto de sete equações na especificação semântica. O exemplo é transcrito a seguir:

type Stack [elementype: Type]

sintaxe

NEWSTACK → Stack

PUSH (Stack, elementtype) → Stack

POP (Stack) → Stack

TOP (Stack) → elementtype U {UNDEFINED}

ISNEW (Stack) → Boolean

REPLACE (Stack, elementtype) → Stack

semântica

declare stk: Stack, elm: elementtype

POP (NEWSTACK) = NEWSTACK

POP (PUSH (stk, elm)) = stk

TOP (NEWSTACK) = UNDEFINED

TOP (PUSH (stk, elm)) = elm

ISNEW (NEWSTACK) = TRUE

ISNEW (PUSH (stk, elm)) = FALSE

REPLACE (stk, elm) = PUSH (POP (stk, elm))

Observe que a expressão elementtype é uma variável pertencente ao conjunto de tipos e varia sobre elementtype. Isto significa que os elementos da Stack são de qualquer tipo (mas todos do mesmo tipo). Segundo Guttag [11], esta especificação não define um único tipo abstrato mas, um "esquema de tipo". Para reduzir o esquema à especificação de um único tipo abstrato, é necessário efetuar a ligação de elementtype a um tipo particular. Por exemplo: Stack [elementtype: Integer] reduz o esquema à especificação de um tipo abstrato particular: uma Stack cujo elementos são inteiros. Uma vez que a transformação de um esquema de tipo para uma especificação de um tipo particu

lar é muito simples, não se fará distinção entre estes dois conceitos durante o transcorrer deste trabalho, convencionando-se que ambos serão denominados "especificação de tipos abstratos".

Para completar o projeto de especificação do tipo abstrato Stack é necessário fornecer uma implementação do tipo. A implementação apresentada, a seguir, é fornecida por Guttag[11] onde cada configuração do Stack é representada por uma estrutura com dois componentes: um array (não limitado) cujos elementos são do tipo elementtype e um inteiro que serve para indicar a posição do elemento do topo da stack no array. O tipo abstrato Array, aqui utilizado foi definido no capítulo anterior.

Implementação do tipo de dados Stack em termos do par (Array, Integer)

representação:

```
STACK (Array [Integer, elementtype], Integer) →
                                    → Stack [elementtype]
```

programas:

```
declare arr: Array, t: Integer, elm: elementtype;
NEWSTACK = STACK (NEWARRY, 0)
PUSH (STACK (arr, t), elm) = STACK (ASSIGN(arr, t+1, elm),
                                                t + 1)
POP (STACK (arr, t)) = IF t = 0
                       THEN STACK (arr, 0)
                       ELSE STACK (arr, t-1)
TOP (STACK (arr, t)) = ACCESS (arr, t)
ISNEW (STCAK (arr, t)) = (t = 0)
```

REPLACE (STACK (arr, t), elm) = IF t = 0
THEN STACK (ASSIGN (arr, 1, elm),1)
ELSE STACK (ASSIGN (arr, t, elm),t)

Para caracterizar as dificuldades existentes na especificação de um tipo abstrato, é necessário um exemplo mais completo que a stack. Um bom exemplo é a especificação do tipo Queue (não limitada) que é uma lista onde os elementos são tratados na forma first-in-first-out, isto é, a ordem da retirada dos elementos é a mesma que a de inserção. A especificação apresentada aqui é similar aquela fornecida por Guttag, Horowitz e Musser [12].

Especificação do tipo abstrato Queue

type Queue [ítem]

sintaxe

NEWQ → Queue

ADDQ (Queue, ítem) → Queue

DELETEQ (Queue) → Queue

FRONTQ (Queue) → ítem U {UNDEFINED}

ISNEWQ (Queue) → Boolean

APPENDQ (Queue, Queue) → Queue

semântica

declare q_1, q_2: Queue, i: ítem

ISNEWQ (NEWQ) = TRUE

ISNEWQ (ADDQ) (q_1,i) = FALSE

DELETEQ (NEWQ) = NEWQ

DELETEQ (ADDQ (q_1,i)) = IF ISNEWQ (q_1)
THEN NEWQ
ELSE ADDQ (DELETEQ(q_1), i)

FRONTQ (NEWQ) = UNDEFINED

FRONTQ (ADDQ (q_1, i)) = IF ISNEWQ (q_1)
THEN i
ELSE FRONTQ (q_1)

APPENDQ (q_1, NEWQ) = q_1

APPENDQ (q_1, ADDQ (q_2, i)) = ADDQ (APPENDQ (q_1,q_2),i)

A complexidade inserida por este exemplo refere-se à recursividade exigida nas operações DELETEQ, FRONTQ e APPENDQ. Como se pode observar, a aplicação da operação FRONTQ a uma Queue vazia resulta em um valor indefinido (FRONTQ(NEWQ)= UNDEFINED); por outro lado, se FRONTQ é aplicada a uma Queue cujo ítem mais recentemente inserido é i e q_1 representa o resto da Queue, então é feito o teste: Se q_1 é vazia o resultado é i, caso contrário, FRONTQ é aplicada recursivamente sobre q_1. Portanto, o resultado de FRONTQ (q_1), onde q_1 não é vazia, é o ítem que ocupa a posição denominada frente da queue.

A mesma situação é encontrada nas operações DELETEQ e APPENDQ uma vez que DELETEQ deve eliminar o elemento que está na frente da Queue e APPENDQ deve unir as duas Queues inserindo no fim da primeira os elementos da segunda, em ordem, a partir do elemento da frente.

Para exemplificar a especificação de um tipo abstrato limitado Guttag, Horowitz e Musser [12] apresentam uma Bqueue de tamanho limitado, nesta especificação algumas restrições impostas à definição dos axiomas para tipos abstratos não limitados são desconsideráveis como, por exemplo, é permitida uma notação que se parece com o uso convencional de procedures e também é relaxada a restrição que todas as operações tenham um úni

co valor. Faz-se também necessária a inclusão de funções auxiliares, também chamadas "hidden functions", para simplificar a especificação, da mesma forma que a introdução de procedures não essenciais podem simplificar e dar maior clareza e um programa. Estas funções são identificadas colocando-se um asterísco na frente da especificação sintática para indicar que não devem ser acessíveis ao usuário, uma vez que elas fazem parte apenas da especificação da abstração e não da abstração do tipo limitado Bqueue.

```
type: Bqueue [item: Type]
sintaxe
   NEWQ(Integer)  →  Bqueue
   *ADDQ(Bqueue, item)→ Bqueue
   *DELETEQ(Bqueue) →  Bqueue
   FRONTQ(Bqueue) → item U {UNDEFINED}
   ISNEWQ(Bqueue) → Boolean
   APPENDQ(Bqueue,Bqueue) →  Bqueue
   SIZE(Bqueue)  → Integer
   LIMIT(Bqueue) →  Integer
   ENQ (var Bqueue, item) →  .
   DEQ (var Bqueue)  → item
semântica
   declare q, r: Bqueue, i: ítem, in: Integer
   ISNEWQ(NEWQ(in)) = true
   ISNEWQ(ADDQ(q,i)) = false
   DELETEQ(NEWQ(in)) = NEWQ (in)
   DELETEQ(ADDQ(q,i)=
          if ISNEWQ(q) then NEWQ (in)
                       else ADDQ (DELETEQ(q),i)
```

```
FRONTQ(NEWQ(in)) = UNDEFINED |underflow|
FRONTQ(ADDQ(q,i))=
        if ISNEWQ (q) then i else FRONTQ(q)
APPENDQ(q,NEWQ(in)) = q
APPENDQ(r,ADDQ(q,i)) = ADDQ(APPENDQ(r,q),i)
LIMIT(NEWQ(in) = in
LIMIT(ADDQ(q,i)) = LIMIT(q)
ENQ(q,i) = if SIZE(q)<LIMIT(q)
                        then q ← ADDQ(q,i)
                        else q ← UNDEFINED [overflow]
DEQ(q) = q ← DELETEQ(q); FRONTQ(q)
SIZE(NEWQ(in)) = 0
SIZE(ADDQ(q,i)) = 1 + SIZE(q)
```

4.2 Um exemplo mais completo - A Grid

O objetivo desta etapa é propiciar uma visão mais ampla na especificação de tipos limitados.

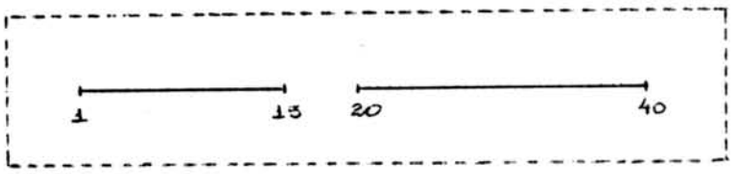
O exemplo escolhido para a especificação, a grid, é adequado pelo fato de que não é um tipo com propriedades conhecidas, e além disto apresenta características que não são encontradas em tipos comumente estudados. O sentido de limitado, na grid, é mais amplo que na Bqueue apresentada na secção anterior pois a partir de sua declaração, a grid tem um número fixo de elementos que é mantido constante durante toda a sua existência, enquanto que na definição do tipo Bqueue, o sentido de limitado refere-se apenas ao número máximo de elementos.

Outros problemas surgirão desde a sua especificação até a sua implementação. Estes problemas serão estudados e soluções serão propostas.

A grid é um estrutura de dados definida por Gehani[4] e consiste em um array que pode ter qualquer formato com a peculiaridade que seus elementos não necessitam estar conectados. Pode ser visualizada como sendo a união de alguns arrays menos a união de outros arrays. Os arrays envolvidos na estrutura de uma grid são chamados componentes.

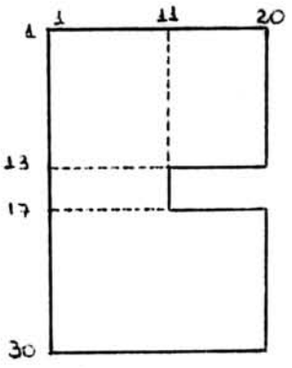
A seguir mostra-se alguns formatos de grids e suas respectivas declarações:

1 - Unidimensional



GRID [1..15 PLUS 20..40] OF real

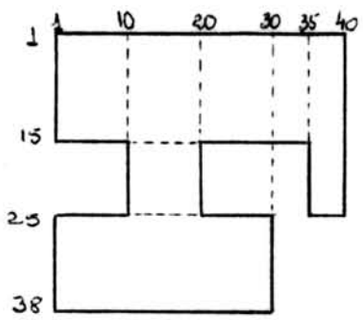
2 - Bidimensional



GRID [1..30, 1..20 MINUS 13..17, 11..20] OF Integer

Como pode ser observado, a expressão PLUS precedendo um array componente, indica a inclusão de seus elementos na grid, enquanto que a expressão MINUS antes de um componente indica que seus elementos não serão considerados elementos da Grid.

Por exemplo, a declaração da Grid cujo formato é apresentado abaixo pode ser dada por



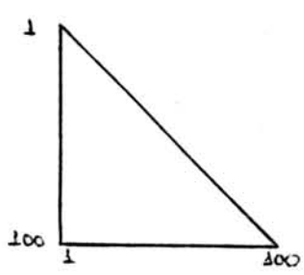

```
GRID [1..38, 1..30 MINUS 16..24, 1..9 PLUS 1..15, 30..35   PLUS
      1..25, 35..40 MINUS 16..24, 21..29]   OF integer
```

ou

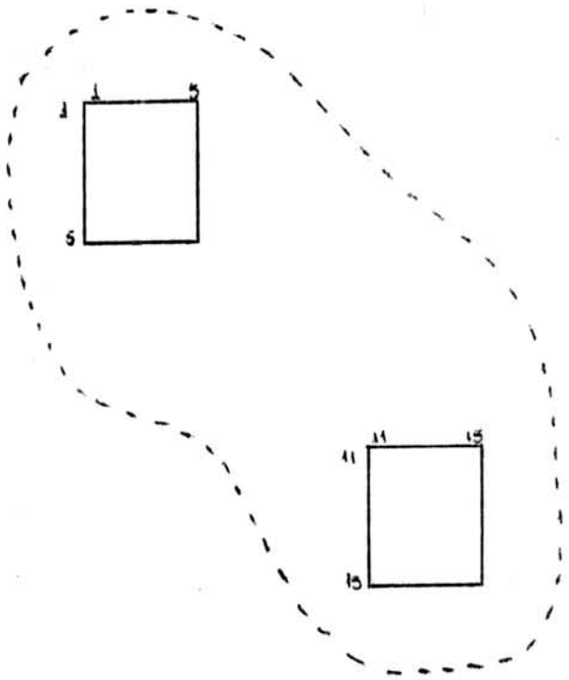
```
GRID [1..38, 1..40 MINUS 16..24, 1..9 MINUS 16..24, 21..34
      MINUS 25..38, 31..35 MINUS 26..38, 35..40]  OF integer
```

ou qualquer outra combinação de PLUS e MINUS que represente apenas os elementos da parte hachuriada.

Outros exemplos:

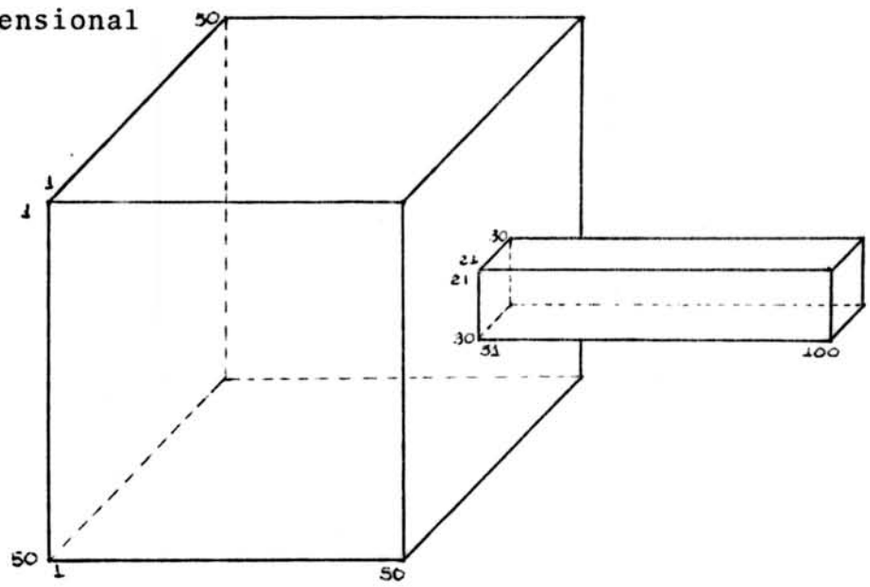



```
GRID [1..1, 1..1 PLUS 1..2, 2..2   PLUS
      1..3, 3..3
      ⋮
      1..99, 99..99 PLUS 1..100, 100..
      100]  OF real
```



GRID [1..5, 1..5 PLUS 11..15, 11..15] OF complex

3 - Tridimensional



GRID [1..50, 1..50, 1..50 PLUS 21..30, 51..100, 21..30]

OF integer

Muitas vezes a declaração de uma grid fica inconveniente como é o caso da declaração da grid triangular (que é a união de 100 arrays lineares).

A declaração de grids pode se tornar mais flexível se permitirmos que os limites da dimensão de um componente sejam funções inteiras de outras coordenadas de dimensão.

Por exemplo, para declararmos a grid triangular faríamos:

GRID [i IN 1..100, 1..i] OF real

Uma grid n-dimensional pode ser pensada como uma função cujo domínio é o conjunto de todos os subscritos válidos.

Seja g uma variável grid de tipo g_t com componentes

q_i $(1 \leq i \leq a)$ e r_j $(1 \leq j \leq b)$

onde todos os rj são prefixados com MINUS. Então o domínio de g

$$dom(g) = dom(g_t) =$$

$$= \bigcup_{i=1}^{a} \begin{matrix}\text{conj. dos subscritos} \\ \text{válidos definidos} \\ \text{pelo componente} \\ q_i\end{matrix} - \bigcup_{j=1}^{b} \begin{matrix}\text{conj. dos subscritos} \\ \text{válidos definidos} \\ \text{pelo componente} \\ r_j\end{matrix}$$

Exemplo:

VAR a: GRID [1..3 PLUS 5..9 MINUS 6..7] OF real

dom(a) = [<1>, <2>, <3>, <5>, <8>, <9>]

Várias aplicações computacionais tais como na Engenharia e análise numérica apresentam problemas decorrentes da necessidade de uma estrutura de dados do tipo grid. Gehani [4] apresenta um programa que ilustra algumas vantagens do uso da grid tais como:

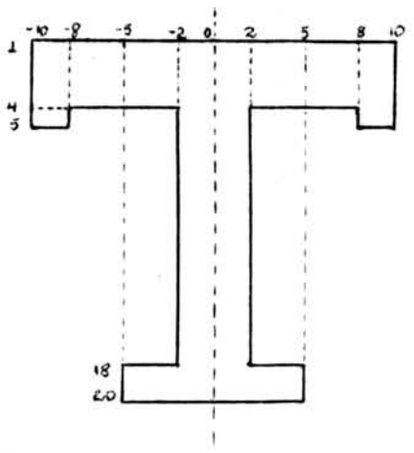
i) a grid reflete claramente o formato a ser representado; e

ii) a alteração na declaração da grid permite que o programa trabalhe sobre qualquer outro formato; e

iii) existe uma economia de memória uma vez que os elementos dos arrays retirados na declaração não ocupam espaço físico.

4.3 Uma Restrição de Grid

O subconjunto escolhido para a especificação será restrito aquelas grids bidimensionais cujo formato possa ser observado a partir da união de retângulos, sejam eles disjuntos ou não, menos a união de outros retângulos. Por exemplo, a configuração da grid apresentada abaixo, pode ser obtida pela união dos retângulos especificados na declaração.



```
GRID [1..3, -10..10 PLUS 4..5, -10.
     ..-8 PLUS 4..5, 8..10 PLUS 4..
     17, -2..2 PLUS 18..20, -5..5]
     OF real

            ou

GRID [1..20, -10..0 MINUS 4..5,  -7
     ..-3 MINUS 6..17, -10..-3
     MINUS 18..20, -10..6 PLUS 1..
     20, 1..10 MINUS 6..17,  3.. 7
     MINUS 6..20, 6..10]  OF real
```

Cada elemento da Grid é representado por um par de inteiros (i,j) onde i representa a posição horizontal e j a posição vertical do elemento. Sendo assim, pode-se verificar que os elementos (2,8), (4,9), (10,1), (19,5) e (20, -3) pertencem à Grid declarada anteriormente enquanto que (5,-6), (10, -4),

(12,6), (19,11) e (20,7) não pertencem.

As operações permitidas sobre uma Grid são atribuição de valor a um elemento e, recuperação do valor de um elemento.

4.4 Considerações sobre a Especificação

De acordo com o que foi apresentado na secção anterior, uma Grid é composta basicamente de uma declaração e de operações de acesso e atribuição de valor aos elementos.

Para que a consistência seja preservada a grid será definida como uma estrutura estática, isto é, uma vez terminada a declaração não será mais permitida a adição ou retirada de elementos.

Basicamente a Grid poderia ser especificada pelas equações

DECLARE (Delgrid) → Grid

ASSIGNG (Grid, elemento, valor) → Grid

ACCESSG (Grid, elemento) → valor U {UNDEFINED}

onde Delgrid representa a declaração dos retângulos pertencentes a Grid, elemento representa o par (i,j) que identifica a sua posição na Grid e valor representa a informação que se deseja atribuir ou acessar.

A operação ACCESSG apresenta dois possíveis resultados um "valor" ou "UNDEFINED". O resultado UNDEFINED é fornecido quando o endereço do elemento acessado não é um endereço válido para a grid, isto é, o elemento não pertence à grid.

Para que a especificação represente realmente o ti-

po que se deseja definir optou-se pela definição de Delgrid como um tipo abstrato que irá descrever as operações realizadas durante a declaração da Grid. Desta forma, os elementos de Delgrid serão retângulos e isto cria a necessidade de especificação de um outro tipo abstrato: o Retanggr. A especificação de Retanggr é extremamente particular já que é apresentada com o único objetivo de auxiliar a definição do tipo Delgrid.

O tipo Retanggr é uma quadrúpla de números inteiros dispostos de tal forma que o primeiro é menor ou igual ao segundo e o terceiro é menor ou igual ao quarto.

Uma especificação para Retanggr é dada logo a seguir da especificação dos tipos Delgrid e Grid.

A especificação dos tipos Delgrid e Grid é fornecida em um único conjunto de axiomas algébricos pelo fato de que a existência de uma Grid irá depender da existência de uma Delgrid. Em outras palavras, somente após o reconhecimento completo de uma declaração de Grid é que serão permitidas atribuições e acessos aos elementos da Grid. Da mesma forma, uma vez terminada a declaração, não serão mais permitidas adições ou retiradas de elementos da Grid. A Grid é, portanto, uma estrutura estática.

Na especificação da Grid, pelo fato dela ser um tipo estático, surgiu a necessidade da definição de duas operações que não devem ser acessíveis ao usuário ("hidden - functions") ISING e ISINDCL. Estas operações devem ser definidas para permitir a verificação da existência ou não de um elemento quando é realizada uma operação ACCESSG ou ASSIGNG. Note-se que ISING e ISINDCL fazem parte da especificação propriamente dita apenas para auxiliar a definição semântica das opera-

ções ACCESSG e ASSIGNG. Uma vez que a Grid é um tipo estático, um ASSIGNG só tem sentido quando realizada sobre um elemento que realmente pertence à Grid.

Observa-se, ainda, que um ASSIGNG pode ser executado sobre uma Delgrid e portanto, deve ser possível verificar a existência do elemento e isto é feito através de ISINDCL.

Para diferenciar as operações que não devem ser acessíveis ao usuário, convenciona-se que elas devem aparecer, na especificação sintática, precedidas por um asterisco (*).

Apresenta-se a seguir a especificação algébrica-axiomática para os tipos Delgrid e Grid, pressupondo-se a existência do tipo Retanggr cuja especificação será dada posteriormente.

4.5 Definições de Grid e Delgrid

type Grid [domaintype: Type x Type, rangetype: Type]
type Delgrid [Retanggr]

sintaxe
DECLARENEWGRID (Retanggr) → Delgrid
DECLAREGPLUS (Delgrid, Retanggr) → Delgrid
DECLAREGMINUS (Delgrid, Retanggr) → Delgrid
ASSIGNG (Delgrid U Grid, domaintype, rangetype) → Grid
ACCESSG (Grid, domaintype) → rangetype U {UNDEFINED}
*ISINDCL (Delgrid, domaintype) → Boolean
*ISING (Grid, domaintype) → Boolean

semântica

declare dcl: Delgrid; gr: Grid; rtg_1: Retanggr;

(i_1,j_1), (i_2,j_2): domaintype, r: rangetype

ACCESSG (ASSIGNG(dcl, (i_1,j_1), r), (i_2,j_2) =

= IF $i_1 = i_2 \quad \Lambda \quad j_1 = j_2$

THEN IF ISINDCL (dcl, (i_1,j_1))

THEN r

ELSE UNDEFINED

ELSE UNDEFINED

ACCESSG (ASSIGNG(gr, (i_1,j_1), r), (i_2,j_2)) =

= IF $i_1 = i_2 \quad \Lambda \quad j_1 = j_2$

THEN IF ISING (gr, (i_1,j_1))

THEN r

ELSE UNDEFINED

ELSE ACCESSG (gr, (i_2,j_2))

ISINDCL (DECLARENEWGRID (rtg_1), (i_1,j_1)) = ISINR (rtg_1, (i_1,j_1))

ISINDCL (DECLAREGPLUS(dcl, rtg_1), (i_1,j_1)) =

= ISINDCL (dcl, (i_1,j_1)) V ISINR (rtg_1, (i_1,j_1))

ISINDCL (DECLAREGMINUS (dcl, rtg_1), (i_1,j_1)) =

= ISINDCL (dcl, (i_1,j_1)) $\Lambda \neg$ ISINR (rtg_1, (i_1,j_1))

ISING (ASSIGNG (dcl, (i_1,j_1), r), (i_2,j_2)) =

= ISINDCL (dcl, (i_2,j_2))

ISING (ASSIGNG (gr, (i_1,j_1), r), (i_2,j_2)) =

= ISING (gr, (i_2,j_2))

Definição de Retanggr

type: Retanggr (abscissainfr: Integer, abscissasupr: Integer,
ordenadainfr: Integer, ordenadasupr: Integer)

sintaxe

CREATERETANG (Integer, Integer, Integer, Integer) → Retanggr

ABSCISSAINFR(Retanggr) → abscissainfr

ABSCISSASUPR(Retanggr) → abscissasupr

ORDENADAINFR(Retanggr) → ordenadainfr

ORDENADASUPR(Retanggr) → ordenadasupr

ISINR (Retanggr, (Integer, Integer)) → Boolean

semântica

declare rtg_1: Retanggr, i_1, j_1, i_2, j_2: Integer

ABSCISSAINFR(CREATERETANG(i_1, i_2, j_1, j_2)) = IF $i_1 \leq i_2$
THEN i_1
ELSE i_2

ABSCISSASUPR(CREATERETANG(i_1, i_2, j_1, j_2)) = IF $i_1 \leq i_2$
THEN i_2
ELSE i_1

ORDENADAINFR(CREATERETANG(i_1, i_2, j_1, j_2)) = IF $j_1 \leq j_2$
THEN j_1
ELSE j_2

ORDENADASUPR(CREATERETANG(i_1, i_2, j_1, j_2)) = IF $j_1 \leq j_2$
THEN j_2
ELSE j_1

ISING (rtg_1, (i_1,j_1)) = ABSCISSAINFR(rtg_1) $\leq i_1 \leq$ ABSCISSASUPR (rtg_1)
$\wedge$ ORDENADAINFR(rtg_1) $\leq j_1 \leq$ ORDENADASUPR (rtg_1)

As operações definidas sobre Retanggr são

CREATERETANG que tem como parâmetros quatro números inteiros e

devolve um Retanggr; as operações ABSCISSAINFR, ABSCISSASUPR, ORDENADAINFR e ORDENADASUPR que recebem um Retanggr e devolvem, cada uma, um dos quatro inteiros usados na criação de Retanggr, e a operação ISINR que tem como parâmetros um Retanggr e um par de inteiros (i, j) e fornece, como resultado, um valor TRUE ou FALSE se o elemento (i, j) é, respectivamente, interior ou exterior ao Retanggr.

4.6 Verificação da "Sufficient-Completeness" da Axiomatização da Grid

É importante verificar se a combinação dos axiomas que definem a semântica da Grid realmente apresentam todas as informações necessárias para o completo entendimento do objetivo das operações do tipo abstrato Grid. Guttag [10] trabalha sobre este tópico e definiu o conceito de suficientemente completo, como foi descrito na secção 3.3.1 deste trabalho.

A definição semântica da Grid e da Delgrid será examinado aqui, a fim de se verificar se estão suficientemente completas.

Sejam: dcl, del: Delgrid, gr, grl: Grid, rtg_1, rtg_2: Retanggr, (i_1, j_1), (i_2, j_2): domaintype, r: rangetype

Seja O o conjunto das operações que executam o mapeamento dos tipos Grid e Delgrid em outro tipo, isto é

O = {ACCESS, ISINDCL, ISING}

Inicialmente estudar-se-á a operação ISINDCL

1) Afirmação: ISINDCL(dcl, (i_2,j_2)) pode se expresso sem conter operações definidas pela especificação dos tipos abstratos Grid e Delgrid.

Prova por indução no número de operações na definição de dcl.

Base de indução: dcl = DECLARENEWGRID (rtg_1)

ISINDCL(DECLARENEWGRID(rtg_1),(i_2,j_2))=

ISINR(rtg_1,(i_2,j_2))

Hipótese de Indução: suponha que

ISINDCL(dcl,(i_2,j_2))

pode ser expresso sem conter operações definidas pela especificação dos tipos abstratos Grid e Delgrid, para qualquer dcl, uma aplicação sucessiva de até n operações dos tipos abstratos Grid e Delgrid.

Passo de Indução: suponha dcl uma aplicação sucessiva de n+1 operações definidas para o tipo abstrato Delgrid.
Dois casos serão considerados.

i) dcl = DECLAREGPLUS(del, rtg_1)

ii) dcl = DECLAREGMINUS(del, rtg_1)

onde del é uma aplicação sucessiva de n operações do tipo abstrato Delgrid, portanto por hipótese de indução ISINDCL(del,(i_2,j_2)) pode ser expresso sem conter operações definidas pela especificação dos tipos abstratos Grid e Delgrid.
Logo:

i) dcl = DECLAREGPLUS(del, rtg_1)

ISINDCL(dcl,(i_2,j_2))=ISINDCL(DECLAREGPLUS(del,rtg_1),(i_2,j_2))

=ISINDCL(del,(i_2,j_2))V ISINR(rtg_1,(i_2,j_2))

ii) dcl = DECLAREGMINUS(del, rtg_1)

ISINDCL(dcl,(i_2,j_2))=ISINDCL(DECLAREGMINUS(del ,rtg_1),
(i_2,j_2))
=ISINDCL(del,(i_2,j_2)) $\wedge$ ISINR(rtg_1,
(i_2,j_2))

Então ISINDCL(dcl,(i_2,j_2)) pode ser expresso sem conter operações definidas pela especificação dos tipos abstratos Grid e Delgrid.

2) Afirmação: ACCESSG(gr,(i_2,j_2)) pode ser expresso sem conter operações definidas pela especificação dos tipos abstratos Grid e Delgrid.

Demonstração:

i) gr=ASSIGNG(del,(i_1,j_1),r)

ACCESSG(ASSIGNG(del,(i_1,j_1)),r),(i_2,j_2))=

IF i_1=i_2 $\wedge$ j_1=j_2

THEN IF ISINDCL(dcl,(i_1,j_1))

THEN r

ELSE UNDEFINED

ELSE UNDEFINED

Pela afirmação 1, segue-se a afirmação 2,

ii) gr=ASSIGNG(gr, (i_1,j_1),r)

Análogo a 1

3) Afirmação:

ISING(gr,(i_2,j_2)) pode ser expresso sem conter operações definidas pela especificação dos tipos abstratos Grid e Delgrid.

Demonstração:

i) gr=ASSIGNG(del, (i_1, j_1), r)

Análogo ao primeiro caso da afirmação 2

ii) gr=ASSIGNG (gr, (i_1, j_1), r)

Análogo a afirmação 1.

Assim fica provado que os axiomas que definem Grid e Delgrid são suficientemente completos.

Os conjuntos de axiomas que definem os outros tipos de dados envolvidos também devem ser provados ser suficientemente completos, entretanto não será provado neste trabalho.

5 IMPLEMENTAÇÃO DA GRID

Como já foi dito anteriormente, os elementos de uma Grid são tratados da mesma forma que os elementos de um Array porém com a característica que os elementos da Grid não estão, necessariamente, ligados uns aos outros como os elementos de um Array. Em outras palavras, a Grid pode ser vista como um conjunto de vários Arrays, disjuntos ou não.

A idéia inicial seria, portanto, implementar a Grid diretamente sobre um Array, porém os problemas surgidos a partir desta concepção, justificam a resolução de se dividir a implementação em três etapas que serão tratadas posteriormente.

5.1 Considerações Gerais

Os tipos abstratos definidos até agora são criados a partir de uma operação que tem como resultado uma configuração vazia do tipo e os elementos são inseridos, na estrutura, na ordem em que aparecem. Mais especificamente, um elemento passa a existir somente após a atribuição de algum valor a ele. A situação é a mesma para os tipos limitados pois, o sentido de "limitado" refere-se apenas ao número máximo de elementos que o tipo pode possuir mas não implica na existência de tal número de elementos. Em outras palavras, os tipos abstratos apresentados são estruturas dinâmicas. A Grid, ao contrário é uma estrutura estática, uma vez que seus elementos passam a existir a partir de sua declaração.

A declaração da Grid foi definida sobre um tipo abs-

trato, denominado Delgrid, que é uma estrutura dinâmica pois os elementos são inseridos ou retirados de acordo com a necessidade que o usuário possa ter quanto ao formato da Grid. Conforme a especificação dada para a Delgrid, seus elementos são do tipo Retanggr e ela é criada a partir da operação DECLARENEWGRID (Retanggr). Também pode ser observado que a Delgrid é um tipo não limitado pois não existem restrições quanto ao número de inserções.

Após a primeira aplicação da operação ASSIGNG não é mais permitido a aplicação de operações de declaração. Isto implica em que só neste momento seja possível fazer a passagem da Delgrid para a Grid.

O problema encontrado na implementação da Grid diz respeito exatamente a esta passagem do tipo dinâmico, não limitado, Delgrid, para o tipo estático e limitado Grid.

5.2 Análise dos Problemas Surgidos e Possíveis Soluções

A seguir serão apresentados os problemas decorrentes da implementação e as soluções adotadas. Estas soluções não são únicas mas foram consideradas pelo fato de simplificarem a tarefa da implementação.

5.2.1 Definição dos Tipos Retang e Point

É necessário identificar cada componente em uma declaração de Grid. Estes componentes são identificados por quatro

números inteiros: i_1, i_2, i_3, i_4 que determinam uma região no espaço bidimensional, Z x Z, limitada pelas retas $y=i_1$, $y=i_2$, $x=i_3$, $x=i_4$. Esta área representa uma região que deve ser adicionada ou retirada da Grid dependendo se a operação é uma operação de PLUS ou MINUS. Para representar abstratamente esta região, optou-se pela definição do tipo abstrato Retang que será criado a partir do fornecimento de quatro números inteiros.

Outro problema diz respeito à referência a um elemento da Grid. Uma vez que a Grid é definida sobre o espaço bidimensional, nada mais natural que representar seus elementos através de pares ordenados (i,j) onde i e j são números inteiros. Define-se, então, um tipo abstrato Point que será formado a partir de dois números inteiros.

São definidas sete operações sobre o tipo Retang, duas sobre o tipo Point e uma sobre Retang e Point. A funcionalidade destas operações é descrita a seguir

Sejam

r_1, r_2, r_3, r_4 inteiros que definem um Retang ret_1

s_1, s_2, s_3, s_4 inteiros que definem um Retang ret_2

i, j inteiros que definem um Point pt

então a operação

ABSCISSAINF(ret_1) fornece como resultado o inteiro r_1

ABSCISSASUP(ret_1) fornece como resultado o inteiro r_2

ORDENADAINF(ret_1) fornece como resultado o inteiro r_3

ORDENADASUP(ret_1) fornece como resultado o inteiro r_4

ISRETANG(ret_1) fornece como resultado o valor TRUE se ret_1 representa um retângulo, isto é, $r_1 \leq r_2$ e $r_3 \leq r_4$ e o valor FALSE caso contrário

$ret_1 \cap ret_2$ tem como resultado um Retang que representa a região de interseção das regiões definidas por ret_1 e ret_2. Caso a interseção seja vazia, o resultado é a quádrupla (1,0,1,0).

$ret_1 - ret_2$ tem como resultado uma lista de Retang's que definem regiões cuja união resulta na região representada por ret_1 da qual se retirou a interseção com a região representada por ret_2.

ABSCISSA (pt) fornece como resultado o inteiro i

ORDENADA (pt) fornece como resultado o inteiro j

ISIN(ret_2,pt) tem como resultado um valor TRUE se pt é interno à região representada por ret_2 e um valor FALSE caso contrário.

Os inteiros que definem um Retang são chamados na ordem em que aparecem, de abscissainf, abscissasup, ordenadainf e ordenadasup.

A especificação para os tipos Retang e Point é dada a seguir:

Definição de Retang e Point

```
type: Retang [abscissainf:Integer, abscissasup:Integer
              ordenadainf:Integer, ordenadasup:Integer]

type: Point  [abscissa: Integer, ordenada: Integer]
```

sintaxe:

ABSCISSAINF(Retang) → abscissainf

ABSCISSASUP(Retang) → abscissasup

ORDENADAINF(Retang) → ordenadainf

ORDENADASUP(Retang) → ordenadasup

ISRETANG(Retang) → Boolean

Retang ∩ Retang → Retang

Retang - Retang → Retanglist

ABSCISSA(Point) → abscissa

ORDENADA(Point) → ordenada

ISIN(Retang,Point) → Boolean

semântica:

declare ret_1, ret_2, rtg_1, rtg_2, rtg_3, rtg_4:Retang; pt:Point

ABSCISSAINF($ret_1 \cap ret_2$) =

= IF ISRETANG(ret_1) ∧ ISRETANG(ret_2)

THEN IF ABSCISSAINF(ret_1) ≤ ABSCISSAINF(ret_2) ≤ ABSCISSASUP(ret_1)

THEN IF ORDENADAINF(ret_1) ≤ ORDENADAINF(ret_2) ≤ ORDENADASUP(ret_1) V ORDENADAINF(ret_2) ≤ ORDENADAINF(ret_1) ≤ ORDENADASUP(ret_2)

THEN ABSCISSAINF(ret_2)

ELSE 1

ELSE IF ABSCISSAINF(ret_2) ≤ ABSCISSAINF(ret_1) ≤ ABSCISSASUP(ret_2)

THEN IF ORDENADAINF(ret_1)$\leq$ORDENADAINF
(ret_2)$\leq$ ORDENADASUP(ret_1) V
ORDENADAINF(ret_2)$\leq$ORDENADAINF
(ret_1)$\leq$ ORDENADASUP(ret_2)
THEN ABSCISSAINF(ret_1)
ELSE 1
ELSE 1
ELSE 1

ABSCISSASUP($ret_1 \cap ret_2$)=
= IF ISRETANG(ret_1)$\wedge$ ISRETANG(ret_2)
THEN IF ABSCISSAINF(ret_1)$\leq$ABSCISSASUP(ret_2)$\leq$ABSCISSASUP
(ret_1)
THEN IF ORDENADAINF(ret_1)$\leq$ORDENADAINF(ret_2)$\leq$
ORDENADASUP(ret_1) V ORDENADAINF(ret_2)$\leq$
ORDENADAINF(ret_1)$\leq$ORDENADASUP(ret_2)
THEN ABSCISSASUP(ret_2)
ELSE 0
ELSE IF ABSCISSAINF(ret_2)$\leq$ABSCISSASUP(ret_1)$\leq$
ABSCISSASUP(ret_2)
THEN IF ORDENADAINF(ret_1)$\leq$ORDENADAINF
(ret_2)$\leq$ORDENADASUP(ret_1) V
ORDENADAINF(ret_2)$\leq$ORDENADAINF
(ret_1)$\leq$ORDENADASUP(ret_2)
THEN ABSCISSASUP(ret_1)
ELSE 0
ELSE 0
ELSE 0

```
ORDENADAINF(ret1 ∩ ret2) =
   = IF ISRETANG(ret1) ∧ ISRETANG(ret2)
        THEN IF ABSCISSAINF(ret1) ≤ABSCISSAINF(ret2) ≤ABSCISSASUP
                (ret1) V ABSCISSAINF(ret2) ≤ABSCISSAINF(ret1) ≤
                ABSCISSASUP(ret2)
                THEN IF ORDENADAINF(ret1) ≤ORDENADAINF(ret2) ≤
                        ORDENADASUP(ret1)
                        THEN ORDENADAINF(ret2)
                        ELSE IF ORDENADAINF(ret2) ≤ORDENADAINF
                                (ret1) ≤ORDENADASUP(ret2)
                                THEN ORDENADAINF(ret1)
                                ELSE 1
                ELSE 1
        ELSE 1

ORDENADASUP(ret1 ∩ ret2) =
   = IF ISRETANG(ret1) ∧ ISRETANG(ret2)
        THEN IF ABSCISSAINF(ret1) ≤ABSCISSAINF(ret2) ≤ABSCISSASUP
                (ret1) V ABSCISSAINF(ret2) ≤ABSCISSAINF(ret1) ≤
                ABSCISSASUP(ret2)
                THEN IF ORDENADAINF(ret1) ≤ORDENADASUP(ret2) ≤
                        ORDENADASUP(ret1)
                        THEN ORDENADASUP(ret2)
                        ELSE IF ORDENADAINF(ret2) ≤ORDENADASUP
                                (ret1) ≤ORDENADASUP(ret2)
                                THEN ORDENADASUP (ret1)
                                ELSE 0
                ELSE 0
        ELSE 0
```

$ISRÉTANG(ret_1)$ = $ABSCISSAINF(ret_1) \leq ABSCISSASUP(ret_1) \wedge$
$ORDENADAINF(ret_1) \leq ORDENADASUP(ret_1)$

$ISIN(ret_1, pt)$ = $ABSCISSAINF(ret_1) \leq ABSCISSA(pt) \leq ABSCISSASUP(ret_1)$
$\wedge ORDENADAINF(ret_1) \leq ORDENADA(pt) \leq ORDENADASUP(ret_1)$

$ret_1 - ret_2$ = IF $ABSCISSAINF(rtg_1) = ABSCISSAINF(ret_1) \wedge ABSCISSASUP(rtg_1) = ABSCISSAINF(ret_2) - 1$
$\wedge ORDENADAINF(rtg_1) = ORDENADAINF(ret_1) \wedge ORDENADASUP(rtg_1) = ORDENADASUP(ret_1)$
$\wedge ABSCISSAINF(rtg_2) = ABSCISSAINF(ret_2) \wedge ABSCISSASUP(rtg_2) = ABSCISSASUP(ret_2)$
$\wedge ORDENADAINF(rtg_2) = ORDENADAINF(ret_2) \wedge ORDENADASUP(rtg_2) = ORDENADAINF(ret_2) - 1$
$\wedge ABSCISSAINF(rtg_3) = ABSCISSAINF(ret_2) \wedge ABSCISSASUP(rtg_3) = ABSCISSASUP(ret_2)$
$\wedge ORDENADAINF(rtg_3) = ORDENADASUP(ret_2) + 1 \wedge ORDENADASUP(rtg_3) = ORDENADASUP(ret_1)$
$\wedge ABSCISSAINF(rtg_4) = ABSCISSASUP(ret_2) + 1 \wedge ABSCISSASUP(rtg_4) = ABSCISSASUP(ret_1)$
$\wedge ORDENADAINF(rtg_4) = ORDENADAINF(ret_1) \wedge ORDENADSUP(rtg_4) = ORDENADASUF(ret_1)$
THEN $INSERT(INSERT(INSERT(NEWLIST(rtg_1), rtg_2), rtg_3), rtg_4)$

5.2.2 - Definição do Tipo Retanglist

Para que se possa saber quais os elementos que per-

tencem à Grid é preciso conhecer toda a declaração, então surge a necessidade de uma estrutura que guarde as informações fornecidas por sucessivas aplicações de operações de declaração. Para este problema, optou-se pela utilização de um tipo abstrato, denominado Retanglist, que irá guardar os limites dos componentes da Grid e a informação de tratamento para cada componente (se aparecem numa operação DECLAREGPLUS ou DECLAREGMINUS).

A Retanglist é uma estrutura formada por uma lista de Retang's precedido por um sinal + ou -. Um Retang precedido pelo sinal + é chamado retângulo positivo enquanto que um Retang precedido pelo sinal - é chamado retângulo negativo. O sinal + precedendo um Retang significa que os pontos internos a este Retang são pontos pertencentes à estrutura, e o sinal - precedendo um Retang significa que os pontos internos a este Retang devem ser retirados da estrutura. Portanto, um ponto pertence a uma estrutura definida por uma Retanglist se é interno a um retângulo positivo e não é interno a um retângulo negativo posterior na lista.

A funcionalidade das operações definidas sobre uma Retanglist é dada a seguir.

Seja ret_1 um Retang, retlist uma Retanglist e i um inteiro

NEWLIST(ret_1): se ret_1 realmente representa um retângulo, isto é, ISRETANG(ret_1)=TRUE, uma Retanglist é criada e nela são inseridos o sinal + seguido dos quatro inteiros que identificam ret_1, caso contrário, a Retanglist é criada vazia.

INSERT(retlist, ret_1): caso ret_1 represente um retângulo, os inteiros que identificam ret_1 são inseridos em retlist precedidos do sinal +, caso contrário, retlist permanece inalterada.

DELETE(retlist, ret_1): Da mesma forma que na operação INSERT, ret_1 será inserido precedido pelo sinal - caso ret_1 represente um retângulo, caso contrário, a retlist permanece inalterada.

ISINL(retlist, pt): devolve um valor TRUE caso pt seja interno a alguma das regiões definidas pelo retlist e um valor FALSE caso contrário.

SIZE(retlist): devolve um valor inteiro que indica o número de Retang's que compõem a Retlist.

RETURN(retlist,i): devolve o i-ésimo Retang inserido na Retanglist se o inteiro i for menor ou igual a SIZE(retlist) e uma mensagem de indefinido caso i > SIZE(retlist).

É necessário verificar se um Retang representa realmente um retângulo porque Retang é definido simplesmente como uma quádrupla de inteiros e as únicas quádruplas que interessam para a formação de uma Grid são aquelas que realmente representam um retângulo, isto é, são Retanggr's. Observe que na operação de interseção de Retang's pode resultar o Retang (1,0,1,0), que não representa um retângulo.

A especificação para o tipo abstrato Retlist é dada a seguir

Definição de Retanglist

type: Retanglist

sintaxe:

NEWLIST(Retang) → Retanglist

INSERT(Retanglist,Retang) → Retanglist

DELETE(Retanglist,Retang) → Retanglist

ISINL(Retanglist,Point) → Boolean

SIZE(Retanglist) → Integer

RETURN(Retanglist,Integer) → Retang U {UNDEFINED}

RETSIG(Retanglist,Integer) → {+, -, UNDEFINED}

semântica

Declare retlist: Retanglist; rtg_1: Retang, pt:Point,i:Integer

ISINL(NEWLIST(rtg_1),pt)=IF ISRETANG(rtg_1)
THEN ISIN(rtg_1,pt)
ELSE FALSE

ISINL(INSERT(retlist,rtg_1),pt)=IF ISRETANG(rtg_1)
THEN ISINL(retlist,pt) V
ISIN(rtg_1,pt)
ELSE ISINL(retlist,pt)

ISINL(DELETE(retlist,rtg_1),pt)=IF ISRETANG(rtg_1)
THEN ISINL(retlist,pt) Λ
¬ ISIN(rtg_1,pt)
ELSE ISINL(retlist,pt)

SIZE(NEWLIST(rtg_1))=IF ISRETANG(rtg_1)
THEN 1
ELSE 0

```
SIZE(INSERT(retlist,rtg1))=IF ISRETANG(rtg1)
                           THEN SIZE(retlist) + 1
                           ELSE SIZE(retlist)
SIZE(DELETE(retlist,rtg1))=IF ISRETANG(rtg1)
                           THEN SIZE (retlist) + 1
                           ELSE SIZE (retlist)
RETURN(NEWLIST(rtg1),i)= IF ISRETANG(rtg1)
                         THEN IF i = 1
                                 THEN rtg1
                                 ELSE UNDEFINED
                         ELSE UNDEFINED
RETURN(INSERT(retlist,rtg1),i)=IF ISRETANG(rtg1)
                               THEN IF i > SIZE(retlist) + 1
                                       THEN UNDEFINED
                                       ELSE IF i =       SIZE
                                                 (retlist) + 1
                                                 THEN rtg1
                                                 ELSE RETURN
                                                  (retlist,i)
                               ELSE RETURN(retlist,i)
RETURN(DELETE(retlist,rtg1),i)=IF ISRETANG(rtg1)
                               THEN IF i > SIZE(retlist) + 1
                                       THEN UNDEFINED
                                       ELSE IF i = SIZE
                                                    (retlist+1)
                                                 THEN rtg1
                                                 ELSE RETURN
                                                  (retlist,i)
                               ELSE RETURN(retlist,i)
```

```
RETSIG(NEWLIST(rtg1),j)=
  = IF ISRETANG(rtg1)
       THEN IF j = 1
                THEN +
                ELSE UNDEFINED
       ELSE UNDEFINED

RETSIG(INSERT(retlist,rtg1),j)=
  = IF ISRETANG(rtg1)
       THEN IF j = SIZE(retlist) + 1
                THEN +
                ELSE IF j > SIZE(retlist) + 1
                         THEN UNDEFINED
                         ELSE RETSIG(retlist,j)
       ELSE IF j > SIZE(retlist)
                THEN UNDEFINED
                ELSE RETSIG(retlist,j)

RETSIG(DELETE(retlist,rtg1),j)=
  = IF ISRETANG(rtg1)
       THEN IF j = SIZE(retlist) + 1
                THEN -
                ELSE IF j > SIZE(retlist) + 1
                         THEN UNDEFINED
                         ELSE RETSIG(retlist,j)
       ELSE IF j > SIZE(retlist)
                THEN UNDEFINED
                ELSE RETSIG(retlist,j)
```

5.2.3 DEFINIÇÃO DO TIPO LIST

Para determinar os elementos que devem pertencer à Grid é necessário percorrer a Retanglist de forma a recuperar cada Retang e a informação referente a cada um, isto é, se a região definida pelo Retang deve ser inserida ou retirada da Grid. Como já foi dito anteriormente, se o Retang é precedido pelo sinal + significa inserção da região e se for precedido pelo sinal - significa retirada.

Para cada Retang ret_1, precedido pelo sinal - a Retanglist deve ser percorrida desde o início até a posição anterior a ret_1 de forma a realizar a retirada da região definida por ret_1 das outras regiões definidas pelos Retang's anteriores a ret_1 na Retanglist. Da mesma forma, cada Retang ret_1 precedido pelo sinal +, com exceção do primeiro, também implica em percorrer a Retanglist de forma a retirar da região definida por ret_1 a área que seja comum às regiões definidas pelos Retang's que precedem ret_1.

O problema aqui identificado é resultante do fato de que as retiradas acima referenciadas são efetuadas através da operação de diferença definida em Retang(Retang-Retang) e esta operação tem como resultado uma lista de Retang's que deve ser inserida na posição do Retang alterado. O tipo abstrato Retanglist não possui operação que permita a inserção de Retang's em uma posição diferente do final da Retanglist. Isto gerou a necessidade de criação de uma estrutura que permitisse a retirada e inserção de elementos em qualquer posição da estrutura. A solução encontrada foi a definição do tipo abstrato List.

A List é uma estrutura cujos elementos podem ser de qualquer tipo (mas todos do mesmo tipo). A List é uma seqüência ordenada de elementos cuja principal característica é que os elementos podem ser inseridos em uma determinada posição estabelecida pela especificação de um valor inteiro i como parâmetro da operação de inserção (INLIST). Da mesma forma, pode-se recuperar o i-ésimo elemento da List, (através da operação ELEMENT) ou simplesmente retirá-lo da List (através da operação DELIST).

A funcionalidade das operações definidas para o tipo abstrato List é dada a seguir.

Sejam lst_1 e lst_2 duas List's, i um inteiro ($i \neq 0$) e elm um Retang.

CREATE: não tem parâmetros e fornece como resultado uma List que inicialmente é vazia.

INLIST (lst_1, i, elm): insere elm em lst_1 na posição i + 1 se existe o i-ésimo elemento.

DELIST (lst_1, i): suprime de lst_1 o elemento que ocupa a i-ésima posição, caso exista o i-ésimo elemento.

JOIN (lst_1, lst_2, i): insere os elementos de lst_2 em lst_1 a partir da posição i + 1.

SIZEL (lst_1): tem como resultado um inteiro que indica o número de elementos pertencentes à lst_1.

ELEMENT (lst_1, i): tem como resultado o elemento que ocupa a i-ésima posição de lst_1. Caso i seja maior que SIZEL(lst_1), o resultado é uma mensagem de UNDEFINED.

A especificação para o tipo abstrato List é dada a seguir:

Definição de List

type: List [elementtype: Type]

sintaxe:

CREATE → List

INLIST (List, Integer, elementtype) → List

DELIST (List, Integer) → List

JOIN (List, List, Integer) → List

SIZEL (List) → Integer

ELEMENT (List, Integer - {0}) → elementtype U {UNDEFINED}

semântica

declare $list_1$, $list_2$: List; i, j; Integer;

elm: elementtype

DELIST(CREATE, i) = CREATE

JOIN($list_1$, CREATE, i) = $list_1$

SIZEL(CREATE) = 0

SIZEL(INLIST($list_1$, i, elm)) = IF i $\leq$ SIZEL($list_1$)
THEN SIZEL($list_1$) + 1
ELSE SIZEL($list_1$)

SIZEL(DELIST($list_1$,i)) = IF i $\leq$ SIZEL($list_1$)
THEN SIZEL($list_1$) - 1
ELSE SIZEL($list_1$)

SIZEL(JOIN($list_1$,$list_2$,i)) = SIZEL($list_1$)+SIZEL($list_2$)

ELEMENT(CREATE, i) = UNDEFINED

ELEMENT(INLIST($list_1$, i, elm), j)=

= IF j > SIZEL(INLIST($list_1$, i, elm))

THEN UNDEFINED

ELSE IF i + 1 = j

THEN elm

ELSE ELEMENT($list_1$,j)

ELEMENT(DELIST($list_1$,i), j)=

= IF j > SIZEL(DELIST($list_1$, i))

THEN UNDEFINED

ELSE IF i = j

THEN ELEMENT($list_1$, i + 1)

ELSE ELEMENT($list_1$, j)

ELEMENT(JOIN($list_1$, $list_2$, i), j)=

= IF j > SIZEL(JOIN($list_1$, $list_2$, i))

THEN UNDEFINED

ELSE IF j $\leq$ i

THEN ELEMENT($list_1$, j)

ELSE IF j $\leq$ i + SIZEL($list_2$)

THEN ELEMENT($list_2$, j - i)

ELSE ELEMENT($list_1$, j -SIZEL($list_2$))

5.2.4 Definição do Tipo Array

O problema agora consiste em se reservar um espaço para os elementos da Grid e ao mesmo tempo possibilitar uma forma de acesso a estes elementos. Esta alocação de espaço só é feita quando o tipo List já está completamente formado, isto é, quando seus elementos definem, univocamente as regiões que for-

mam a Grid. Os elementos da List, neste caso específico, são do tipo Retang e como se sabe, um Retang é definido por quatro números inteiros. Estes quatro números inteiros serão utilizados agora para identificar cada componente da Grid.

O número de elementos de um componente será calculado a partir dos limites do componente, de acordo com a expressão:

$$n = (ABSCISSASUP(rtg_1) - ABSCISSAINF(rtg_1)+1)*(ORDENADASUP(rtg_1) - ORDENADAINF(rtg_1) + 1)$$

onde

rtg_1 é um Retang pertencente à List e $ABSCISSAINF(rtg_1)$, $ABSCISSASUP(rtg_1)$, $ORDENADAINF(rtg_1)$ e $ORDENADASUP(rtg_1)$ são respectivamente, os quatro números que fornecem os limites de um componente representado por rtg_1.

A alocação de espaço será representada através da criação de um Array unidimensional cujos elementos serão acessados por um valor inteiro (índice). Neste Array serão colocados os limites, obtidos, a partir da recuperação dos elementos da List, na seguinte ordem: os quatro números que limitam um componente e o número de elementos deste componente. A seguir são deixadas tantas posições livres quantos forem os elementos deste componente para, só então, inserir os limites e o número de elementos do próximo componente.

Guttag, Horowitz e Musser [11] apresentam uma especificação para um Array não limitado, cujos componentes são de qualquer tipo. Esta especificação já foi apresentada, em partes, no capítulo 3 deste trabalho.

A abstração é fornecida, agora, de forma completa:

type: Array [domaintype: Type, rangetype: Type]

sintaxe:

NEWARRAY → Array

ASSIGN(Array, domaintype, rangetype) → Array

ACCESS(Array, domaintype) → rangetype U {UNDEFINED}

semântica:

declare arr: Array; d_1, d_2; domaintype: r: rangetype

$$ACCESS(NEWARRAY, d_1) = UNDEFINED$$

$$ACCESS(ASSIGN(arr,d_1,r),d_2) = \text{IF } d_1 = d_2 \text{ THEN } r \text{ ELSE } ACCESS(arr,d_2).$$

5.2.5 Funções Recursivas

Examinando-se as implementações de tipos abstratos existentes na bibliografia, observa-se que elas são simples, pois, no caso geral, cada tipo abstrato é implementado sobre um outro tipo abstrato de nível mais baixo. Em outras palavras, existe uma hierarquia de modelos para a implementação de tipos abstratos. Já na implementação da Grid apresentada isto não ocorre pois ela é implementada sobre vários tipos abstratos e assume diversas formas no decorrer da implementação. Mais especificamente: na fase de declaração, a Grid é implementada como uma Retanglist, quando a primeira operação ASSIGNG é executada, a Grid toma a forma de uma List e logo a seguir, de um Array. Para efetuar esta transição é necessário percorrer o tipo anterior para retirar informações que permitam a criação do próximo. É nes-

te aspecto que o tipo abstrato Grid apresenta a dificuldade de implementação.

A implementação mostra como a semântica é realizada na máquina e portanto pode ser vista como um programa. Para facilitar a prova de correção de uma implementação, prefere-se fornecer esta implementação em termos de outros tipos abstratos que se supõe estejam implementados. Portanto, o programa que implementa um tipo abstrato deve ser escrito seguindo as regras de uma linguagem de especificação de tipos abstratos.

A linguagem de especificação utilizada foi definida no capítulo 3 e as únicas facilidades de controle apresentadas são: o comando IF-THEN-ELSE e a utilização de funções recursivas.

No programa que implementa a Grid, faz-se necessária a utilização de uma estrutura de controle do tipo do comando FOR do Algol, isto é, um comando que permita a repetição de um procedimento um número determinado de vezes. Por exemplo, na passagem da Retanglist para List uma das etapas consiste em percorrer a Retanglist examinando-se cada um de seus elementos e efetuando-se uma série de procedimentos para cada elemento examinado. Uma vez que não existe um comando FOR que permita este tipo de ação na linguagem utilizada, e como qualquer procedimento efetivo pode ser definido através de funções recursivas, a solução encontrada foi a utilização de funções definidas recursivamente.

Um exemplo fácil da definição de um procedimento efetivo através de funções recursivas é apresentado a seguir:

```
ISINL(RETLIST(arr, i), pt) =
  = IF i = 0
       THEN FALSE
       ELSE IF ACCESS(arr, i - 4) = -
              THEN IF ACCESS(arr, i - 3) ≤ ABSCISSA(pt) ≤ ACCESS
                     (arr, i - 2) ∧ ACCESS(arr, i - 1) ≤ORDENADA
                     (pt) ≤ ACCESS(arr,i)
                     THEN FALSE
                     ELSE ISINL(RETLIST(arr, i - 5), pt)
              ELSE IF ACCESS(arr, i - 3) ≤ ABSCISSA(pt) ≤ ACCESS
                     (arr, i - 2) ∧ ACCESS(arr, i - 1) ≤ORDENADA
                     (pt) ≤ACCESS(arr, i)
                     THEN TRUE
                     ELSE ISINL(RETLIST(arr, i - 5), pt)
```

Este procedimento consiste na implementação da operação ISINL definida para o tipo abstrato Retanglist. Aqui, arr é um Array, i é um Integer que representa o tamanho de arr e pt é um Point.

O processo executa a busca de pt em uma Retanglist que é representada por RTLIST(arr, i).

5.3 A Implementação Propriamente Dita

Conforme descrito na secção anterior, a implementação do tipo abstrato Grid requer a utilização dos tipos abstratos Point, Retang, Retanglist, List e Array. Com exceção do tipo

abstrato Array, que é considerado um tipo primitivo, todos os tipos envolvidos na implementação da Grid devem possuir, também, uma implementação. Esta implementação será fornecida logo após a implementação da Grid.

Examinando-se a especificação do tipo Grid, apresentada no capítulo anterior, pode-se notar que ela é dividida em duas partes lógicas: a parte da declaração, onde se utiliza o tipo abstrato Delgrid, e a parte das operações de acesso e atribuição onde já é utilizado o tipo abstrato Grid. Da mesma forma, a implementação deverá preocupar-se com estas duas etapas e, ainda, com uma etapa intermediária que trata da passagem do tipo Delgrid para o tipo Grid.

5.3.1 A GRID E A DELGRID

representação: GRD(Array, Integer) → Grid
DCL(Retanglist) → Delgrid

programas:

declare: retlist: Retanglist, rtg_1: Retang, rtgr: Retanggr
arr: Array, i, k: Integer, (i_1,j_1): domaintype
r: rangetype

G1) DECLARENEWGRID(rtgr) = DCL(NEWLIST(RTG(ABSCISSAINFR (rtgr),
ABSCISSASUPR(rtgr),
ORDENADAINFR(rtgr),
ORDENADASUPR(rtgr)))

G2) DECLAREGPLUS(DCL(retlist), rtgr) = DCL(INSERT(retlist,RTG
(ABSCISSAINFR(rtgr),
ABSCISSASUPR(rtgr),
ORDENADAINFR(rtgr),
ORDENADASUPR(rtgr))))

G3) DECLAREGMINUS(DCL(retlist), rtgr) =DCL(DELETE(retlist, RTG
(ABSCISSAINFR(rtgr),
ABSCISSASUPR(rtgr),
ORDENADAINFR(rtgr),
ORDENADASUPR(rtgr))))

G4) ISINDCL(DCL(retlist), (i_1,j_1))=ISINL(retlist, (i_1,j_1))

G5) ASSIGNG(DCL(retlist), (i_1,j_1),r)=

```
IF ISINL(retlist, (i1,j1))
   THEN IF SIZE(retlist) = 1
           THEN GRD(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN
                  (ASSIGN(NEWARRAY,
                  1,ABSCISSAINFR(RETURN(retlist,1))),
                  2,ABSCISSASUPR(RETURN(retlist,1))),
                  3,ORDENADAINFR(RETURN(retlist,1))),
                  4,ORDENADASUPR(RETURN(retlist,1))),
                  5, (ABSCISSASUPR(RETURN(retlist,1))-
                  ABSCISSAINFR(RETURN(retlist,1))+1)*
                  (ORDENADASUPR(RETURN(retlist,1))-
                  ORDENADAINFR(RETURN(retlist,1))),
                  (j1-ORDENADAINFR(RETURN(retlist,1))+1)*
                  (i1-ABSCISSAINFR(RETURN(retlist,1))+1)+
                  i1-ABSCISSAINFR(RETURN(retlist,1))+  6,
                                                  r),5)
```

```
                ELSE ASSIGNG(GRD(RESERVM(DOLIST(retlist)),
                                SIZEARR(DOLIST(retlist))),(i1,j1),r)
        ELSE GRD(RESERVM(DOLIST(retlist))),SIZEARR(DOLIST
                                                (retlist)))
```

G6) ASSIGNG(GRD(arr,i), (i_1,j_1)) =

```
        = IF SCANG(arr,i,i1,j1,1) = UNDEFINED
                                THEN FALSE
                                ELSE TRUE
```

G7) ASSIGNG(GRD(arr,i), (i_1,j_1),r)=

```
        = IF SCANG(arr,i,i1,j1,1) = UNDEFINED
             THEN GRD(arr,i)
             ELSE GRD(ASSIGN(arr,SCANG(arr,i,i1,j1,1)+  (j1-
                    ACCESS(arr,SCANG(arr,i,i1,j1,1)+2  +1)*
                    (i1-ACCESS(arr,SCANG(arr,i,i1,j1,1))+1)
                    + (i1-ACCESS(arr,SCANG(arr,i,i1,j1,1))+
                    1 + 4, r), i)
```

G8) ACCESSG(GRD(arr,i), (i_1,j_1))=

```
        = IF SCANG(arr, i,i1,j1) = UNDEFINED
             THEN UNDEFINED
             ELSE ACCESS(arr,SCANG(arr,i,i1,j1,1)+(j1-ACCESS
                    (arr,SCANG(arr,i,i1,j1,1)+2)+1)*(i1-
                    ACCESS(arr,SCANG(arr,i,i1,j1,1))+1)+
                    (i1-ACCESS(arr,SCANG(arr,i,i1,j1,1))
                    +1+4)
```

Descrição das funções auxiliares da implementação da Grid

1) RTLISTTOLIST(Retanglist, Integer$^+$) → List

Seja retlist uma Retanglist e i um Integer$^+$

RTLISTTOLIST(retlist, i) = IF i = 1

THEN INLIST(CREATE, 1,RETURN

(retlist,1))

ELSE INLIST(RTLISTTOLIST

(retlist,i), i+1, RETURN

(retlist, i + 1))

Esta função forma uma List, a partir de retlist, inserindo um a um os i+ 1 elementos de retlist na List que é inicialmente vazia.

2) DELINTFIRST(Retanglist, List, Integer$^+$, Integer$^+$) →

(Retanglist, List, Integer$^+$, Integer$^+$)

Seja retlist uma Retanglist, lst uma List e j, k, dois Integer$^+$'s

DELINTFIRST(retlist, lst, k, j) =

= (retlist,JOIN(lst,RTLISTTOLIST(RETURN(retlist,j)

- RETURN(retlist,j) ∩ RETURN(retlist,1)),4),k), k +

SIZEL(RTLISTTOLIST(RETURN(retlist,j) - (RETURN

(retlist, j)∩ RETURN(retlist,1)),4), j - 1)

Esta função insere na posição k + 1 de lst a List resultante da operação de diferença entre o j-ésimo e o primeiro Retang de retlist. O endereço para a próxima inserção (k), fica alterado para k + tamanho da List inserida. O próximo Retang a ser recuperado para a comparação com o primeiro é aquele que ocupa a posição j - 1 da retlist.

3) DELINTOTHERS(List, Integer$^+$, Integer$^+$) → List

Seja lst uma List e i, j dois Integer$^+$'s.

DELINTOTHERS(lst, j, i) =

= IF i = 0

THEN lst

ELSE DELIST(JOIN(DELINTOTHERS(lst, j, i - 1),

RTLISTTOLIST(ELEMENT(DELINTOTHERS(lst, j, i -

1), j + i) - (ELEMENT(DELINTOTHERS(lst, j, i -

1), j + i)∩ ELEMENT(DELINTOTHERS(lst, j, i -1),

j),4), j + i), j+ i)

Esta função retira de cada elemento de lst, desde o (j+1)-ésimo até o (j+i)-ésimo, sua intersecção com o j-ésimo elemento e devolve uma List onde os elementos das posições de (j+1) até (j+i) foram retirados e em seus lugares foram inseridas List's obtidas da diferença entre cada elemento e sua intersecção com o j-ésimo elemento.

4) DOMINUS(List, $Integer^+$, $Integer^+$) → List

Seja lst uma List e j, i dois $Integer^+$'s com i<j e j>1

DOMINUS(lst, j, i) = IF i ≤ 1

THEN lst

ELSE DELIST(JOIN(DOMINUS(lst, j, i -

1), RTLISTTOLIST(ELEMENT(DOMINUS

(lst,j, i - 1), i - 1) -(ELEMENT

(DOMINUS(lst, j, i - 1), i - 1)

∩ ELEMENT(DOMINUS(lst, j, i -

1), j)), 4), i - 1), i - 1)

A função DOMINUS(lst, j, i) devolve uma List onde foram inseridas, nas posições de 2 até i as List's formadas pela aplicação de RTLISTTOLIST sobre cada uma das Retanglist's obtidas da operação de diferença entre o k-ésimo elemento (k = 1, 2, ...

i - 1) e sua intersecção com o j-ésimo elemento. Cada k-ésimo elemento (k = 1, 2, ... i - 1) é retirado de lst.

5) SCANLIST(List, Integer$^+$) → List

Seja lst uma List e j um Integer$^+$

SCANLIST(lst, j) = IF j ≤ 1
THEN lst
ELSE SCANLIST(DELINTOTHERS(lst, j - 1,
SIZEL(lst) - j - 2), j - 1)

Esta função percorre lst aplicando a função DELINTOTHERS sobre cada elemento de lst, desde a segunda posição até a posição SIZEL(lst) - j - 2.

6) SCANRETLIST(Retanglist, List, Integer$^+$, Integer$^+$) →
(Retanglist, List, Integer$^+$, Integer$^+$)

Seja retlist uma Retanglist, lst uma List e k, j dois Integer$^+$'s.

SCANRETLIST(retlist, lst, k, j)=
= IF RETSIG(retlist, j) = +
THEN DELINTFIRST(retlist, lst, 1, j)
ELSE (retlist, DOMINUS(lst, j, SIZEL(lst)+1),
k, j)

Esta função examina o sinal do j-ésimo elemento de retlist. Caso o sinal seja +, aplica-se a função DELINTFIRST sobre o j-ésimo elemento, caso contrário, a função DOMINUS é aplicada sobre este elemento.

7) FIRSTVERLIST(Retanglist, Integer$^+$) → List

Seja retlist uma Retanglist e j um Integer$^+$

```
FIRSTVERLIST(retlist, j) = IF j = 1
                           THEN INLIST(CREATE, 0,   RETURN
                                (retlist, 1))
                           ELSE SCANRETLIST(retlist,
                                FIRSTVERLIST(
                                retlist, j - 1),1,j)
```

Esta função cria uma List e insere na primeira posição, o primeiro elemento de retlist. A seguir, percorre retlist, examinando o sinal de cada elemento através da função SCANRETLIST.

8) DOLIST(Retanglist) → List

Seja retlist uma Retanglist

```
DOLIST(retlist) = SCANLIST(FIRSTVERLIST(retlist, SIZE(
                           retlist)),SIZEL(FIRSTVERLIST
                           (retlist, SIZE(retlist))))
```

Esta função fornece, a partir de retlist, uma List cujos elementos são dois a dois disjuntos.

9) RM(List, $Integer^+$, Array, $Integer^+$) →
(List, $Integer^+$, Array, $Integer^+$)

Seja lst uma List, arr um Array e k, j dois $Integer^+$'s

```
RM(lst, arr, j) = IF j > 0
                  THEN RM(lst, k+5 + (ABSCISSASUP(ELEMENT
                       (lst,j)) -    ABSCISSAINF(ELEMENT
                       (lst,j))+1)* (ORDENADASUP(ELEMENT
                       (lst,j) - ORDENADAINF(ELEMENT(lst,
                       j))+1), ASSIGN(ASSIGN(ASSIGN(
                       ASSIGN(ASSIGN(arr,k,   ABSCISSAINF
```

```
                (ELEMENT(lst,j))),
                k+1,ABSCISSASUP(ELEMENT(lst,j))),
                k+2,ORDENADAINF(ELEMENT(lst,j))),
                k+3,ORDENADASUP(ELEMENT(lst,j))),
                k+4,(ABSCISSASUP(ELEMENT(lst,j))-
                ABSCISSAINF(ELEMENT(lst,j))+1)*
                (ORDENADASUP(ELEMENT(lst,j))-
                ORDENADAINF(ELEMENT(lst,j))+1), j-
                1)
        ELSE (lst, k+4, ASSIGN(arr,k+4,0),0)
```

A função RM(lst,k,arr,j) insere em arr, a partir da posição k, os quatro inteiros que identificam cada um dos j elementos de lst, seguidos de um valor calculado pela expressão

```
(ABSCISSASUP(ELEMENT(lst,j))-ABSCISSAINF(ELEMENT(lst,j))+1) *
(ORDENADASUP(ELEMENT(lst,j))-ORDENADAINF(ELEMENT(lst,j)) + 1)
```

10) $\pi 2$ (List, Integer$^+$, Array, Integer$^+$) $\rightarrow$ Integer$^+$

Seja lst uma List, arr um Array e i, j Integer$^+$'s

$\pi 2$ (lst, i, arr, j) = i

Esta função tem como resultado um Integer$^+$ que é o segundo elemento da quádrupla.

11) $\pi 3$ (List, Integer$^+$, Array, Integer$^+$) $\rightarrow$ Array

Seja lst uma List, arr um Array e i, j Integer$^+$'s

$\pi 3$ (lst, i, arr, j) = arr

Esta função tem como resultado um Array, que é o terceiro elemento da quádrupla.

12) CREATEARG(List) → (List, Integer$^+$, Array, Integer$^+$)

Seja lst uma List

CREATEARG(List) = (lst, 1, NEWARRAY, SIZEL(lst))

Esta função devolve, a partir de lst, a própria lst o valor 1, um Array vazio e o tamanho de lst.

13) SIZEARR(List) → Integer$^+$

Seja lst uma List

SIZEARR(lst) = $\pi 2$(RM(CREATEARG(lst)))

Esta função devolve o tamanho do Array criado para conter informações sobre os elementos de lst.

14) RESERVM(List) → Array

Seja lst uma List

RESERVM(lst) = $\pi 3$(RM(CREATEARG(lst)))

Esta função devolve um Array que contém as informações sobre os elementos de lst.

5.3.2 O RETANGGR

O tipo Retanggr será implementado sobre o tipo Retang pois todos os elementos de Retanggr são também elementos de Retang. Note que a recíproca não é verdadeira já que o tipo Retang permite elementos que não satisfazem a alguns axiomas da definição de Retanggr.

representação

RGR(Integer, Integer, Integer, Integer) → Retanggr

programas:

declare i_1, i_2, i_3, i_4, i, j: Integer

CREATERETANG(i_1, i_2, i_3, i_4) =

= IF $i_1 \leq i_2$

THEN IF $i_3 \leq i_4$

THEN RGR(i_1, i_2, i_3, i_4)

ELSE RGR(i_1, i_2, i_4, i_3)

ELSE IF $i_3 \leq i_4$

THEN RGR(i_2, i_1, i_3, i_4)

ELSE RGR(i_2, i_1, i_4, i_3)

5.3.3 A LIST

O tipo abstrato List será implementado sobre o tipo Array, considerando que seus elementos são do tipo Retang.

representação:

LST (Array [Integer, Retang], Integer) → List

programas:

declare: arr, arr_1: Array, rtg_1: Retang, i, j, k:Integer

L1) CREATE = LST(NEWARRAY, 0)

L2) INLIST(LST(arr,i),j,rtg_1)=

= IF j = SIZEL(LST(arr,i))

THEN LST(ASSIGN(ASSIGN(ASSING(ASSIGN(ASSIGN(ASSIGN

(arr,i+5,i+6), i, i+1), i+1, ABSCISSAINF

(rtg_1)), i+2, ABSCISSASUP(rtg_1)), i + 3,

ORDENADAINF(rtg_1)), i+4, ORDENADASUP

(rtg_1)), i+5)

ELSE IF j > SIZEL(LST(arr,i))

THEN LST(arr,i)

```
                    ELSE LST(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN
                        (ASSIGN(arr,i+5,ACCESS(arr,    NEXT
                        (arr, i,j-1)+4)),NEXT(arr,i,j-1)  +
                        4, i+1),
                        i+1,ABSCISSAINF(rtg1)),
                        i+2,ABSCISSASUP(rtg1)),
                        i+3,ORDENADAINF(rtg1)),
                        i+4,ORDENADASUP(rtg1)), i+5)

L3) DELIST(LST(arr, i), j)=
      = IF j > SIZEL(LST(arr,i)) V j = 0
           THEN LST(arr, i)
           ELSE LST(ASSIGN(arr,NEXT(arr,i,j-2)+4,ACCESS(arr,
                NEXT(arr,i,j-1)+4),i)

L4) SIZEL(LST(arr,i))=DOSIZE(LST(arr,i),0)

L5) ELEMENT(LST(arr,i),j)=
      = IF SIZEL(LST(arr,i),j) = 0
           THEN UNDEFINED
           ELSE IF j > SIZEL(LST(arr,i))
                THEN UNDEFINED
                ELSE RTG(ACCESS(arr,NEXT(arr,i,j-1)),
                         ACCESS(arr,NEXT(arr,i,j-1)+1),
                         ACCESS(arr,NEXT(arr,i,j-1)+2),
                         ACCESS(arr,NEXT(arr,i,j-1)+3))

L6) FRONT(CREATE) = 0

L7) FRONT(INLIST(LST(arr,i),j,rtg1))
     = IF j ≠ 0 THEN FRONT(LST(arr,i))
                ELSE i + 1
```

```
L8) FRONT(DELIST(LST(arr,i),j))=
      = IF j ≠ 1  THEN FRONT(LST(arr,i))
                  ELSE ACCESS(arr,FRONT(LST(arr,i))+4)

L9) FRONT(JOIN(LST(arr,i),LST(arr,j),k)=
      = IF k = 0 THEN FRONT(LST(arr1,j))
                 ELSE FRONT(LST(arr,i))

L10) JOIN(LST(arr,i),LST(arr1,j),k)=
      = IF k ≥ SIZEL(LST(arr,i))
          THEN LST(LINK(ASSIGN(ASSIGN(arr,i+SIZEL(arr,j))*5,i+
                 SIZEL(LST(arr1,j))*5+1),NEXT(arr,i,SIZEL(LST
                 (arr,i))-1)+4,i+1),i,arr1,j,(SIZEL(LST(arr1,
                 j))-1)*5),i+SIZEL(LST(arr1,j)))
          ELSE IF SIZEL(LST(arr1,j))=0
               THEN LST(arr,i)
               ELSE LST(ASSIGN(ASSIGN(LINK(arr,i,arr1,j,(SIZEL
                      (LST(arr,j))-1)*5),i+SIZEL(LST(arr1,j))
                      *5, NEXT(arr,i,k))),NEXT(arr,i,k-1)+ 4,
                      i+1)

L11) NEXT(arr,i,j) = IF i = 0
                       THEN 1
                       ELSE IF j < DIV(i,5)
                              THEN IF j = 0
                                     THEN FRONT(LST(arr,i))
                                     ELSE ACCESS(arr,NEXT
                                                 (arr,i,j-1)+4)
                              ELSE IF j = DIV(i,5)
                                     THEN i + 1
                                     ELSE 0
```

L12) DOSIZE(LST(arr,i),j)= IF NEXT(arr,i,j) = i + 1
THEN j
ELSE DOSIZE(LST(arr,i), i + 1)

L13) ATRIB(arr,i,arr_1,j,k)=
= ASSIGN(ASSIGN(arr,i+SIZEL(LST(arr_1,j))*5,NEXT(arr,i,
k)),NEXT(arr,i,k-1)+4, i+ 1)

L14) LINK(arr,i,arr_1,j,m) = IF m $\leq$ 0
THEN INSERTELM(arr,i,arr_1,j,0)
ELSE INSERTELM(LINK(arr,i,arr_1,j,
m-5),i,arr_1,j,m)

L15) INSERTELM(arr,i,arr_1,j,m) = ASSIGN(ASSING(ASSIGN(ASSIGN(
(ASSIGN(arr,
i+m+1,ACCESS(arr_1,NEXT(arr_1,j,DIV(m,5))),
i+m+2,ACCESS(arr_1,NEXT(arr_1,j,DIV(m,5))+1),
i+m+3,ACCESS(arr_1,NEXT(arr_1,j,DIV(m,5))+2),
i+m+4,ACCESS(arr_1,NEXT(arr_1,j,DIV(m,5))+3),
i+m+5, i+m+6)

5.3.4 A RETANGLIST, O RETANG E O POINT

representações:

RTLIST(Array[Integer,elementtype],Integer) → Retanglist

RTG(Integer,Integer,Integer,Integer) → Retang

PT(Integer,Integer) → Point

programas:

declare arr, arr_1: Array; r_1,r_2,r_3,r_4,s_1,s_2,s_3,s_4,i,j:Integer;
pt: Point

RT1) NEWLIST(RTG(r_1,r_2,r_3,r_4))=IF ISRETANG(RTG(r_1,r_2,r_3,r_4))
THEN RTLIST(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(
NEWARRAY,1,+),2,r_1),3,r_2),4,r_3),5,r_4),5)
ELSE RTLIST(NEWARRAY,0)

RT2) INSERT(RTLIST(arr,i),RTG(r_1,r_2,r_3,r_4))=IF ISRETANG(RTG(r_1,
r_2,r_3,r_4))
THEN RTLIST(ASSIGN(ASSIGN(ASSIGN(ASSING(ASSING(arr,i
+1,+),i+2,r_1),i+3,r_2),i+4,r_3),i+5,r_4),i+5)
ELSE RTLIST(arr,i)

RT3) DELETE(RTLIST(arr,i),RTG(r_1,r_2,r_3,r_4))=IF ISRETANG(RTG(r_1,
r_2,r_3,r_4))
THEN RTLIST(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSING(arr,i
+1,-),i+2,r_1),i+3,r_2),i+4,r_3),i+5,r_4),i+5)
ELSE RTLIST(arr,i)

RT4) ISINL(RTLIST(arr,i),pt)=
= IF i = 0
THEN FALSE
ELSE IF ACCESS(arr,i-4)=-
THEN IF ACCESS(arr,i-3)≤ABSCISSA(pt)≤
ACCESS(arr,i-2)∧ ACCESS(arr,i-1)
≤ORDENADA(pt)≤ ACCESS(arr,i)
THEN FALSE
ELSE ISINL(RTLIST(arr,i-5),pt)
ELSE IF ACCESS(arr,i-3)≤ABSCISSA(pt) ≤
ACCESS(arr,i-2)∧ ACCESS(arr,i-
1)≤ ORDENADA(pt)≤ACCESS(arr,i)
THEN TRUE
ELSE ISINL(RTLIST(arr,i-5),pt)

RT5) $RTG(r_1,r_2,r_3,r_4) \cap RTG(s_1,s_2,s_3,s_4) =$

```
1    = IF ISRETANG(RTG(r1,r2,r3,r4)) ∧ ISRETANG(RTG(s1,s2,s3,s4))
2         THEN IF r1 ≤ s1 ≤ r2
3                   THEN IF r1 ≤ s2 ≤ r2
4                             THEN IF r3 ≤ s3 ≤ r4
5                                       THEN IF r3 ≤ s4 ≤ r4
6                                                 THEN RTG(s1,s2,s3,s4)
7                                                 ELSE RTG(s1,s2,s3,r4)
8                                       ELSE IF s3 ≤ r3 ≤ s4
9                                                 THEN IF s3 ≤ r4 ≤ s4
10                                                      THEN RTG(s1,s2,
                                                                 r3,r4)
11                                                      ELSE RTG(s1,s2,
                                                                 r3,r4)
12                                                ELSE RTG(1,0,1,0)
13                            ELSE IF r3 ≤ s3 ≤ r4
14                                      THEN IF r3 ≤ s4 ≤ r4
15                                                THEN RTG(s1,r2,s3,s4)
16                                                ELSE RTG(s1,r2,s3,r4)
17                                      ELSE IF s3 ≤ r3 ≤ s4
18                                                THEN IF s3 ≤ r4 ≤ s4
19                                                      THEN RTG(s1,r2,
                                                                 r3,r4)
20                                                      ELSE RTG(s1,r2,
                                                                 r3,s4)
21                                                ELSE RTG(1,0,1,0)
22                  ELSE IF s1 ≤ r1 ≤ s2
23                            THEN IF s1 ≤ r2 ≤ s2
24                                      THEN IF r3 ≤ s3 ≤ r4
```

```
25                                  THEN IF r3 ≤ s4 ≤ r4
26                                       THEN RTG(r1,r2,
                                                 s3,s4)
27                                       ELSE RTG(r1,r2,
                                                 s3,r4)
28                                  ELSE IF s3 ≤ r3 ≤ s4
29                                       THEN IF s3 ≤ r4
                                                 ≤ s4
30                                            THEN
                                            RTG(r1,r2,r3,r4)
31                                            ELSE
                                            RTG(r1,r2,r3,s4)
32                                       ELSE RTG(1,0,1,0)
33                             ELSE IF r3 ≤ s3 ≤ r4
34                                  THEN  IF r3 ≤ s4≤  r4
35                                       THEN RTG(r1,s2,
                                                 s3,s4)
36                                       ELSE RTG(r1,s2,
                                                 s3,r4)
37                                  ELSE IF s3 ≤ r3 ≤ s4
38                                       THEN IF s3 ≤ r4
                                                 ≤ s4
39                                            THEN
                                            RTG(r1,s2,r3,r4)
40                                            ELSE
                                            RTG(r1,s2,r3,s4)
41                                       ELSE RTG(1,0,1,0)
42                        ELSE RTG(1,0,1,0)
43              ELSE RTG(1,0,1,0)
```

RT6) RTG(r_1,r_2,r_3,r_4) - RTG(s_1,s_2,s_3,s_4)=

= IF ISRETANG(RTG(s_1,s_2,s_3,s_4))

THEN INSERT(INSERT(INSERT(INSERT(NEWLIST(RTG(r_1,s_1 -

1,r_3,r_4)),RTG(s_1,s_2,r_3,s_3-1)), RTG(s_1,s_2,

s_4+1,r_4)),RTG(s_2+1,r_2,r_3,r_4))

ELSE NEWLIST(RTG(r_1,r_2,r_3,r_4))

RT7) ABSCISSAINF(RTG(r_1,r_2,r_3,r_4)) = r_1

RT8) ABSCISSASUP(RTG(r_1,r_2,r_3,r_4)) = r_2

RT9) ORDENADAINF(RTG(r_1,r_2,r_3,r_4)) = r_3

RT10) ORDENADASUP(RTG(r_1,r_2,r_3,r_4)) = r_4

RT11) ISRETANG(RTG(r_1,r_2,r_3,r_4)) = ($r_1 \leq r_2 \wedge r_3 \leq r_4$)

RT12) ISIN(RTG(r_1,r_2,r_3,r_4),PT(i_1,j_1))=

= $r_1 \leq i_1 \leq r_2 \quad \wedge \quad r_3 \leq j_1 \leq r_4$

RT13) ABSCISSA(PT(i_1,j_1)) = i_1

RT14) ORDENADA(PT(i_1,j_1)) = j_1

RT15) SIZE(RTLIST(arr,i)) = DIV(i,5)

RT16) RETURN(RTLIST(arr,j),i) =

= IF i > DIV(j,5)

THEN UNDEFINED

ELSE IF i = DIV(j,5)

THEN RTG(ACCESS(arr,j-3),ACCESS(arr,j -

2),ACCESS(arr,j-1),ACCESS(arr ,

j))

ELSE RETURN(RTLIST(arr,j-5),i)

RT17) RETSIG(RTLIST(arr,i),j)=

= IF j ≤ 0 V j * 5 > i

THEN UNDEFINED

ELSE ACCESS(arr, j*5-4)

5.4 CORREÇÃO DA IMPLEMENTAÇÃO

A correção da implementação consiste em provar que a representação das operações de um tipo abstrato em termos de outros tipos abstratos exibe todas as propriedades esperadas pela especificação do tipo.

A técnica de prova básica para verificar a implementação de um tipo abstrato, definido axiomaticamente, consiste em mostrar que a especificação para o tipo é satisfeita quando os axiomas são substituídos pelos programas correspondentes as operações.

Mais especificamente, dada uma especificação algébrica-axiomática de um tipo de dados T e uma implementação expressa em termos de outros tipos abstratos de dados, os quais também possuem especificações algébricas-axiomáticas, a prova da correção desta implementação pode ser alcançada se, para cada axioma de T, forem considerados os seguintes passos:

- Substituir cada ocorrência de T pela sua representação;
- Considerar a seguinte igualdade:
  lado esquerdo do axioma= implementação do lado esquerdo do axioma.
- via uma série de reduções, usando as propriedades da igualdade (principalmente simetria e transitividade) e o princípio da in

dução. chegar à implementação do lado direito do axioma e consequentemente. ao lado direito.

Estes passos são úteis para se chegar a uma prova de correção. Entretanto, não podem ser diretamente transformados em regras mecanizáveis.

É importante provar a correcão de uma implementação porque nesta fase os erros. que passaram desapercebidos durante a especificação e implementação,são detectados.

Para provar a correção da implementação de um tipo de dados que tenha sido implementado em função de outros tipos abstratos, é necessário discorrer apenas sobre as especificações destes outros tipos e não sobre suas implementações. Portanto uma prova de correção é fatorada em níveis que correspondem aos níveis de implementação. Logo, para demonstrar a correção da Grid basta considerar apenas as especificações dos tipos Point, Retang, Retanglist, List e Array sem preocupar-se com suas implementações.

Para exemplificar a prova de correção considere-se o axioma

ISINDCL(DECLAREGPLUS(dcl,rtgr),(i_1,j_1))=

= ISINDCL(dcl(i_1,j_1)) V ISINR(rtgr, (i_1,j_1))

apresentado na especificação dos tipos abstratos Grid e Delgrid onde dcl é uma Delgrid, rtgr é um Retanggr e (i_1,j_1) é um pt.

Mostra-se que este axioma é satisfeito, partindo do lado esquerdo da igualdade e usando os programas que implementam o tipo, chega-se a uma expressão equivalente a implementação do lado direito da igualdade do axioma. Então, supondo que existe uma Retanglist

retlist tal que dcl = DCL(retlist), vem

ISINDCL(DECLAREGPLUS(dcl,rtgr),(i_1,j_1)) =

= ISINDCL(DECLAREGPLUS(DCL(retlist),rtgr),(i_1,j_1)) (por def.)

= ISINDCL(DCL(INSERT(retlist,RTG(ABSCISSAINFR(rtgr),
ABSCISSASUPR(rtgr),ORDENADAINFR(rtgr),ORDENADASUPR
(rtgr)))), (i_1,j_1)) (progr. G2)

= ISINL(INSERT(retlist,RTG(ABSCISSAINFR(rtgr),ABSCISSASUPR
(rtgr),ORDENADAINFR(rtgr),ORDENADASUPR(rtgr))),(i_1,j_1))
(progr. G4)

= ISINL(retlist,(i_1,j_1)) V ISIN(RTG(ABSCISAINFR(rtgr),
ABSCISSASUPR(rtgr),ORDENADAINFR(rtgr),ORDENADASUPR
(rtgr)),(i_1,j_1)) (pela def. ISINL e ISRETANG(RTG(
ABSCISSAINFR(rtgr),ABSCISSASUPR(rtgr),ORDENADAINFR
(rtgr),ORDENADASUPR(rtgr))=TRUE)

= ISINL(retlist,(i_1,j_1)) V (ABSCISSAINFR(rtgr) $\leq i_1 \leq$
ABSCISSASUPR(rtgr) $\wedge$ ORDENADAINFR
(rtgr) $\leq j_1 \leq$ ORDENADASUPR(rtgr)
(prog. RT12)

= ISINDCL(DCL(retlist),(i_1,j_1)) V ISINR(rtgr,(i_1,j_1)) (prog.
G4 e definição de ISINR em Retanggr)

= ISINDCL(dcl,(i_1,j_1)) V ISINR(rtgr,(i_1,j_1)) (por def.)

Pode-se observar que nesta prova de correção usa-se o axioma ISINL(INSERT(retlist,rtg_1),(i_1,j_1)) apresentado na especificação do tipo Retanglist onde rtg_1 é um Retang. A prova da correção deste axioma é feita em um nível mais baixo e não é necessário apresentá-lo novamente nesta fase.

Da mesma forma, assume-se as igualdades:

ISIN (RTG(ABSCISSAINFR(rtgr),ABSCISSASUPR(rtgr);ORDENADAINFR
(rtrg),ORDENADASUPR(rtgr)), (i_1,j_1))=

= ABSCISSAINFR(rtgr) $\leq i_1 \leq$ ABSCISSASUPR(rtgr) $\wedge$
ORDENADAINFR(rtgr) $\leq j_1 \leq$ ORDENADASUPR(rtgr)

= ISINR(rtgr,(i_1,j_1))

A prova da correção da implementação dos tipos Grid e Delgrid é apresentada no apêndice A e as provas da correção da implementação dos tipos abstratos utilizados na implementação da Grid e Delgrid é apresentada no apêndice B.

5.5 A Utilização do Array na Implementação dos Tipos Abstratos

Na teoria dos tipos abstratos, o Array é considerado um tipo primitivo, isto é, considera-se que já esteja implementado na linguagem de programação. Neste caso, cada operação definida sobre o Array pode ser vista como um programa diretamente executável na máquina. Uma aplicação sucessiva destas operações também irá consistir em um programa.

Conforme já apresentado, os tipos abstratos Retang, Retanglist e List foram implementados sobre o Array e portanto suas operações podem ser vistas como uma aplicação sucessiva das operações válidas sobre o Array.

O tipo abstrato Retanggr é implementado sobre o Retang, o tipo abstrato Delgrid, sobre o Retang, Retanglist e List e a Grid sobre o Array. Portanto, uma vez que a implementação é provada correta, pode-se afirmar que todas as operações dos ti-

pos abstratos citados acima são executáveis pela máquina. Por este motivo, é importante provar a correção da implementação.

Convém notar, porém, que o Array aqui definido é um tipo ilimitado enquanto num Array de uma linguagem de programação é limitado. Portanto, na passagem da especificação abstrata para uma implementação através de uma linguagem de programação, este aspecto deve ser considerado.

6 CONCLUSÃO

Neste trabalho foram apresentadas algumas técnicas para especificação de tipos abstratos dando-se um enfoque especial para a técnica algébrica-axiomática definida por Guttag.

A técnica algébrica-axiomática foi escolhida para exemplificar a abstração de dados por duas razões:

a) - por ser natural expressar tipos abstratos (conjunto de valores e operações) através de uma álgebra, que é também um conjunto de valores e operações, e porque axiomas expressam com facilidade o comportamento e as relações que caracterizam operações e classes de valores.

b) - pela facilidade proporcionada para a prova da correção da implementação, por conter conceitos e formalismos matemáticos bem definidos.

Confirmou-se a afirmação de que tipos abstratos limitados apresentam maior dificuldade para a especificação e que os problemas aumentam quando o tipo limitado é estático. Isto pode ser verificado pelo estudo da Grid que possibilitou a exibição de dificuldades que eram desconhecidas. Foram apresentadas soluções para todos os problemas encontrados embora deva-se ressaltar que estas soluções não são únicas.

Na especificação da Grid surgiu a necessidade de inclusão de funções que não devem ser acessíveis ao usuário "hidden functions" uma vez que não expressam uma operação válida sobre o tipo abstrato, mas são necessárias para clarificar o funcionamento das outras operações.

Surgiram também na especificação da Grid definições paralelas. Entretanto, mostrou-se que a definição de mais

de um tipo em uma mesma especificação, nem sempre aumenta as dificuldades e a complexidade da axiomatização.

Outra preocupação que mostrou ser desnecessária diz respeito à duplicidade de elementos que pertençam à intersecção de componentes da declaração de uma Grid. A exclusão de elementos repetidos já está implicita na relação entre as operações ACCESSG e ASSIGNG. Quando um acesso é feito, apenas o último valor atribuído ao elemento é que retorna, não interessando as atribuições anteriores. Portanto, não tem sentido a existência de dois valores para o mesmo elemento o que obriga a implementação a preocupar-se com este fato.

O fato da Grid ser um tipo estático gerou alguns problemas na implementação. Estes problemas exigiram a utilização de mais de um tipo abstrato e funções recursivas bem complexas (apresentadas no capítulo 5).

Conforme descrito na secção 5.4, existem etapas bem definidas a serem seguidas durante o desenvolvimento de uma prova de correção. Entretanto estas regras não fornecem meios suficientes para uma mecanização do processo. Todavia, as etapas apresentadas, podem ser exploradas a fim de se conseguir uma maior automatização. Outros requisitos seriam necessários tais como linguagem de programação adequada e técnicas de verificação de teoremas.

A Grid apresentada é uma restrição da original. O estudo dos problemas gerados pela definição mais completa origina um bom trabalho na área.

Esta área ainda apresenta muitos tópicos a serem explorados, já que a maioria dos estudos efetuados são baseados em exemplos muito simples (estruturas de dados mais complexas

certamente trarão problemas ainda não considerados).

Espera-se que este trabalho seja ponto de partida para o desenvolvimento de outros trabalhos na área de especificação, implementação e prova de correção.

APÊNDICE 1 - CORREÇÃO DA IMPLEMENTAÇÃO DA GRID

Correção da Implementação da Grid

ACCESSG (ASSIGNG (dcl, (i_1,j_1),r),(i_2,j_2)) =

= ACCESSG(ASSIGNG (dcl (retlist), (i_1,j_1),r),(i_2,j_2)

Suponha que ISINL (retlist, (i_1,j_1)) = TRUE e SIZE (retlist)= 1

Então

ASSIGNG (DCL(retlist),(i_1,j_1) r)= GRD(arr_1,k), onde

arr_1 = ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(NEWARRAY,

1, ABSCISSAINFR (RETURN(retlist,1)),

2, ABSCISSASUPR (RETURN(retlist,1)),

3, ORDENADAINFR (RETURN(retlist,1)),

4, ORDENADASUPR (RETURN(retlist,1)),

5, (ABSCISSASUPR(return(retlist,1)) - ABSCISSAINFR(RETURN

(retlist,1)) + 1) * (ORDENADASUPR(RETURN(retlist,1))-

ORDENADAINFR(RETURN(retlist,1)) + 1)),

(j_1-ORDENADAINFR(RETURN(retlist,1)) + 1) *

(i_1 - ABSCISSAINFR(RETURN(retlist,1)) + 1) + i_1 -

ABSCISSAINFR(RETURN(retlist,1) + 6, r)

e k = 5 (prog G5)

ACCESSG (ASSIGNG(DCL(retlist), (i_1,j_1),r),(i_2,j_2)) =

= ACCESSG(GRD(arr_1,5), (i_2,j_2)) (por definição)

Suponha $i_1=i_2$ $j_1=j_2$ então SCANG(arr_1,5,i_2,j_2,1)=1

pois ISINL(retlist,(i_1,j_1)) = TRUE e SIZE (retlist) = 1 $\Longrightarrow$

ABSCISSAINFR(RETURN(retlist,1))$\leq i_1 \leq$ABSCISSASUPR(RETURN(retlist,

1)) $\wedge$

ORDENADAINFR(RETURN(retlist,1))$\leq j_1 \leq$ORDENADASUPR(RETURN(retlist,

1))

então ACCESS(arr_1,1) $\leq i_2 \leq$ ACCESS(arr_1,2) $\wedge$

ACCESS(arr_1,3) $\leq j \leq$ ACCESS(arr_1,4) (def. SCANG)

logo,

ACCESSG(GRD(arr_1,5),(i_2,j_2))=

=ACCESS(arr_1,SCANG(arr_1,5,i_2,j_2,1)+(j_2-ACCESS(arr_1, SCANG(arr_1, 5,i_2,j_2,1)+2)+1)*(i_2-ACCESS(arr_1, SCANG(arr_1,5,i_2,j_2,1)) + 1) +(i_2- ACCESS(arr_1,SCANG(arr_1,5,i_2,j_2,1))+1+4)

= ACCESS (arr_1,1+(j_2-ACCESS(arr_1,1+2)+1)*(i_2-ACCESS(arr_1,1)+1)+ (i_2-ACCESS(arr_1,1)+1+4)

(pois SCANG(arr_1,5.i_2.j_2,1)=1)

= ACCESS (arr_1,j_2-ORDENADAINFR(RETURN(retlist,1))+1) * (i_2 - ABSCISSAINFR(RETURN(retlist,1))+1)+i_2 - ABSCISSAINFR (RETURN(retlist,1)+6)

= r (* 1)

Se ISINL(retlist,(i_1,j_1)) = TRUE e SIZE (retlist) ≠ 1

tem-se

ASSIGNG (DCL(retlist),(i_1,j_1),r)=

= ASSIGNG(GRD(RESERVM(DOLIST(retlist)),SIZEARR(DOLIST (retlist))),(i_1,j_1),r) (prog G5)

faça: arr_1=RESERVM(DOLIST(retlist))

i=SIZEARR(DOLIST(retlist))

então se i_1=i_2 ∧ j_1=j_2 tem-se

ACCESSG(ASSIGNG(DCL(retlist),(i_1,j_1),r),(i_2,j_2))=

=ACCESSG(ASSIGNG(GRD(arr_1,i),(i_1,j_1),r),(i_2,j_2))

Observe que SCANG(arr_1,i,i_1,j_1,1) = UNDEFINED já que ISINL (retlist,(i_1,j_1)) = TRUE e arr_1=RESERVM(DOLIST(retlist)) e i = SIZEARR(DOLIST(retlist)) e como i_1=i_2 ∧ j_1=j_2 tem-se SCANG (arr_1,i,i_2,j_2,1) ≠ UNDEFINED

Portanto

ASSIGNG(GRD(arr_1,i),(i_1,j_1),r) = GRD(arr_2,i)

onde arr_2 = ASSIGN(arr_1,SCANG(arr_1,i,i_1,j_1,1)+(j_1-ACCESS(arr_1, SCANG(arr_1,i,i_1,j_1,1)+2)+1)*(i_1-ACCESS(arr_1, SCANG (arr_1,i,i_1,j_1,1))+1)+(i_1-ACCESS(arr_1,SCANG(arr_1, i, i_1,j_1,1))+1+4,r)

Por outro lado,

ACCESSG(GRD(arr_2,i),(i_2,j_2)) = ACCESSG(GRD(arr_2,i),(i_1,j_1))

= ACCESS(arr_2,SCANG(arr_2,i,i_1,j_1,1)+(j_1-ACCESS(arr_2, SCANG(arr_2,i,i_1,j_1,1)+2)+1)*(i_1-ACCESS(arr_2,SCANG (arr_2,i,i_1,j_1,1))+1)+(i_1-ACCESS(arr_2,SCANG (arr_2, i,i_1,j_1,1))+1+4)

Observe que

SCANG(arr_2,i,i_1,j_1,1)=SCANG(arr_1,i,i_1,j_1,1)

então

ACCESSG(ASSIGNG(GRD(arr_1,i),(i_1,j_1),r),(i_2,j_2)) =

ACCESS(ASSIGN(arr_1,SCANG(arr_1,i,i_1,j_1,1)+(j_1-ACCESS(arr_1,SCANG (arr_1,i,i_1,j_1,1)+2)+1)*(i_1-ACCESS(arr_1,SCANG(arr_1, i,i_1,j_1,1))+1)+(i_1-ACCESS(arr_1,SCANG(arr_1,i,i_1,j_1, 1))+1+4,r),SCANG(arr_1,i,i_1,j_1,1)+(j_1-ACCESS(arr_2, SCANG(arr_1,i,i_1,j_1,1)+2)+1)*(i_1-ACCESS(arr_2, SCANG (arr_1,i,i_1,j_1,1))+1)+(i_1-ACCESS(arr_2,SCANG(arr_1, i,i_1,j_1,1))+1+4) = r

pela definição de array e pelo fato que

ACCESS(arr_1,SCANG(arr_1,i,i_1,j_1,1)+k)=ACCESS(arr_2,SCANG(arr_1, i, i_1,j_1,1)+k)

Se $0 \leq k \leq 4$, $k \in \mathbb{Z}$, já que estas posições não são alteradas (* 2)

Suponha agora, que ISINL(retlist(i_1,j_1)=FALSE e que

$i_1=i_2 \quad \Lambda \quad j_1=j_2$

Então

$$ASSIGNG(DCL(retlist),(i_1,j_1),r)=$$

$$= GRD(RESERVM(DOLIST(retlist)),SIZEARR(DOLIST(retlist)))$$

Observe que apenas efetuou-se a transformação da retlist em um array com a respectiva reserva de espaço para seus elementos, isto é, nenhum elemento que já não pertencesse à retlist foi incluído no array. Portanto (i_1,j_1) não irá pertencer ao array uma vez que não pertencia à qualquer elemento do retlist.

Chame

$$arr_1 = RESERVM(DOLIST(retlist))$$

$$i = SIZEARR(DOLIST(retlist))$$

isto é,

$$ASSIGNG(DCL(retlist),(i_1,j_1),r) = GRD(arr_1,i)$$

então

$$ACCESSG(ASSIGNG(DCL(retlist),(i_1,j_1),r),(i_2,j_2))=$$

$$= ACCESSG(GRD(arr_1,i),(i_2,j_2))$$

Temos $SCANG(arr_1,i,i_2,j_2,1) = UNDEFINED$ pois

$$i_1=i_2 \quad \Lambda \quad j_1=j_2$$ e portanto

$$ACCESSG(GRD(arr_1,i),(i_2,j_2)) = UNDEFINED \qquad (* 3)$$

Analogamente, se $ISINL(retlist,(i_1,j_1))=$ FALSE mas

$$i_1\neq i_2 \quad V \quad j_1\neq j_2$$ tem-se

$$ASSIGNG(DCL(retlist),(i_1,j_1),r) =$$

$$= GRD(RESERVM(DOLIST(retlist),SIZEARR(DOLIST(retlist))))$$

Conforme já foi salientado anteriormente, a única tarefa da função RESERVM é reservar espaço no array para os e-

lementos da List formada por DOLIST e SIZEARR devolve o tamanho do array formado. Em nenhum momento é efetuado algum ASSIGN sobre os elementos pertencentes a área reservada aos elementos da Grid. Logo, se arr_1= RESERVM(DOLIST(retlist)) e i = SIZEARR (DOLIST(retlist)) tem-se SCANG(arr_1,i,i_2,j_2,1)=UNDEFINED então

ACCESSG(ASSIGNG(DCL(retlist),(i_1,j_1),r),(i_2,j_2))=

= ACCESSG(GRD(arr_1,i),(i_2,j_2)) (* 4)

= UNDEFINED (programa G8)

Então, por *1, *2, *3 e *4 tem-se que

ACCESSG(ASSIGNG(dcl,(i_1,j_1),r),(i_2,j_2)=

IF i_1 = i_2 Λ j_1 = j_2
THEN IF ISINDCL(dcl,(i_1,j_1))
THEN r (*1 e *2)
ELSE UNDEFINED (*3)
ELSE UNDEFINED (*4)

ACCESSG(ASSIGNG(gr,(i_1,j_1),r),(i_2,j_2)=ACCESSG(ASSIGNG(GRD (arr, i),(i_1,j_1),r),(i_2,j_2))

Suponha i_1 = i_2 Λ j_1 = j_2 e que ISING(gr,(i_1,j_1))= TRUE

então

ACCESSG(ASSIGNG(GRD(arr,i),(i_1,j_1),r),(i_2,j_2))=

= ACCESSG(GRD(ASSIGNG(arr,SCANG(arr,i,i_1,j_1,1)+(j_1-ACCESS(arr, SCANG(arr,i,i_1,j_1,1)+2)+1)*(i_1-ACCESS(arr, SCANG (arr,i,i_1,j_1,1))+1)+(i_1-ACCESS(arr,SCANG(arr, i, i_1,j_1,1))+1+4,r),i),(i_2,j_2)) (por G7 e porque SCANG(arr,i,i_1,j_1) ≠ UNDEFINED

já que ISING(GRD(arr,i),(i_1,j_1))=TRUE)

Faça arr_1=ASSIGN(arr,SCANG(arr,i,i_1,j_1,1)+(j_1-ACCESS(arr,SCANG(arr,i,i_1,j_1,1)+2)+1)*(i_1-ACCESS(arr, SCANG(arr,i,i_1,j_1,1))+1+(i_1-ACCESS(arr, SCANG(arr, i,i_1,j_1,1))+1+4,r)

então

ACCESSG(ASSIGNG(GRD(arr,i),(i_1,j_1),r),(i_2,j_2))=

= ACCESSG(GRD(arr_1,i),(i_2,j_2))

= ACCESSG(GRD(arr_1,i),(i_1,j_1)) $(i_1=i_2 \wedge j_1=j_2)$

=ACCESS(arr_1,SCANG(arr_1,i,i_1,j_1,1)+(j_1-ACCESS(arr_1,SCANG(arr_1,i,i_1,j_1,1)+2)+1)*(i_1-ACCESS(arr_1,SCANG(arr_1,i,i_1,j_1,1))+1+(i_1-ACCESS(arr_1,SCANG(arr_1,i,i_1,j_1,1))+1+4)

(prog. G8)

Observe que SCANG(arr_1,i,i_1,j_1,1)=SCANG(arr,i,i_1,j_1,1) e que ACCESS(arr_1,SCANG(arr_1,i,i_1,j_1,1)+k)=ACCESS(arr,SCANG(arr,i,i_1,j_1,1)+k) para $0 \leq k \leq 4$, $k \in \mathbb{Z}$

Então, pela definição de array, tem-se

ACCESS(ASSIGN(arr,SCANG(arr,i,i_1,j_1,1)+(j_1-ACCESS(arr, SCANG(arr,i,i_1,j_1,1)+2)+1)*(i_1-ACCESS(arr,SCANG(arr,i,i_1,j_1,1))+1+(i_1-ACCESS(arr,SCANG(arr,i,i_1,j_1,1))+1+4,r),SCANG(arr_1,i,i_1,j_1,1)+(j_1-ACCESS(arr_1, SCANG(arr_1,i,i_1,j_1,1)+2)+1)*(i_1-ACCESS(arr_1,SCANG(arr_1,i,i_1,j_1,1))+1+(i_1-ACCESS(arr_1,SCANG(arr_1,i,i_1,j_1,1))+1+4)=

= r (* 1)

Suponha

$i_1 = i_2 \wedge j_1 = j_2$ mas ISING(gr,(i_1,j_1))=FALSE

ACCESSG(ASSIGNG(gr,(i_1,j_1),r),(i_2,j_2))=

ACCESSG(ASSIGNG(GRD(arr,i),(i_1,j_1),r),(i_2,j_2))

tem-se

SCANG(arr,i,i_1,j_1,1)=UNDEFINED (pois ISING(GRD(arr,i),(i_1,j_1))=
FALSE)

então

ASSIGNG(GRD(arr,i),(i_1,j_1),r)=GRD(arr,i) (progr. G7)

e

ACCESSG(GRD(arr,i),(i_2,j_2))=UNDEFINED pois

SCANG(arr,i,i_2,j_2,1)=UNDEFINED ($i_1=i_2$ $\wedge$ $j_1=j_2$)

então

ACCESSG(ASSIGNG(gr,(i_1,j_1),r),(i_2,j_2))=UNDEFINED (* 2)

Suponha, agora que $i_1 \neq i_2$ V $j_1 \neq j_2$

Se SCANG(arr,i,i_1,j_1,1)=UNDEFINED tem-se

ASSIGNG(GRD(arr,i),(i_1,j_1),r)=GRD(arr,i) (progr. G7)

e portanto,

ACCESSG(ASSIGNG(GRD(arr,i),(i_1,j_1),r),(i_2,j_2))=

= ACCESSG(GRD(arr,i),(i_2,j_2))

= ACCESSG(gr,(i_2,j_2)) (*3) (por definição)

Se SCANG(arr,i,i_1,j_1,1) = UNDEFINED, segue que

ASSIGNG(GRD(arr,i),(i_1,j_1),r)=GRD(arr_1,i) onde

arr_1=ASSIGN(arr,SCANG(arr,i,i_1,j_1,1)+(j_1-ACCESS(arr,SCANG(arr,
i,i_1,j_1,1)+2)+1)*(i_1-ACCESS(arr,SCANG(arr,i,i_1,j_1,
1))+1+(i_1-ACCESS(arr,SCANG(arr,i,i_1,j_1,1))+1+ '4,
r) (prog. G7)

então

ACCESSG(ASSIGNG(GRD(arr,i),(i_1,j_1),r),(i_2,j_2))=

$= ACCESSG(GRD(arr_1,i),(i_2,j_2))$

Observe que se $SCANG(arr_1,i,i_2,j_2,1)$=UNDEFINED então

$SCANG(arr,i,i_2,j_2,1)$ = UNDEFINED e tem-se

$ACCESSG(GRD(arr_1,1),(i_2,j_2))=ACCESSG(GRD(arr,i),(i_2,j_2))$ (*3)

por outro lado, se $SCANG(arr_1,i,i_2,j_2,1) \neq$ UNDEFINED, tem-se

$ACCESSG(GRD(arr_1,1),(i_2,j_2))=$

$=ACCESS(arr_1,SCANG(arr_1,i,i_2,j_2,1)+(j_2-ACCESS(arr_1,SCANG(arr_1,i,i_2,j_2,1)+2)+1)*(i_2-ACCESS(arr_1,SCANG(arr_1,i,i_2,j_2,1))+1)+(i_2-ACCESS(arr_1,SCANG(arr_1,i,i_2,j_2,1))+1+4)$

Observe que arr_1 é o array resultante da atribuição do valor r ao elemento (i_1,j_1) e como $i_1 \neq i_2$ V $j_1 \neq j_2$ tem-se

$ACCESS(arr_1,SCANG(arr_1,i,i_2,j_2,1)+k)=ACCESS(arr,SCANG(arr,i,i_2,j_2,1)+k)$ como $0 \leq k \leq 4$

Então $ACCESSG(ASSIGNG(GRD(arr,i),(i_1,j_1),r),(i_2,j_2))=$

$=ACCESS(ASSIGNG(arr,SCANG(arr,i,i_1,j_1,1)+(j_1-ACCESS(arr,SCANG(arr,i,i_1,j_1,1)+2)+1)*(i_1-ACCESS(arr,SCANG(arr,i,i_1,j_1,1))+1)+(i_1-ACCESS(arr,SCANG(arr,i,i_1,j_1,1))+1+4,r),SCANG(arr,i,i_2,j_2,1)+(j_2-ACCESS(arr,SCANG(arr,i,i_2,j_2,1)+2+1)*(i_2-ACCESS(arr,SCANG(arr,i,i_2,j_2,1))+1+(i_2-ACCESS(arr,SCANG(arr,i,i_2,j_2,1))+1+4)$

$=ACCESS(arr,SCANG(arr,i,i_2,j_2,1)+(i_2-ACCESS(arr,SCANG(arr,i,i_2,j_2,1)+2+1)*(i_2-ACCESS(arr,SCANG(arr,i,i_2,j_2,1))+1+(i_2-ACCESS(arr,SCANG(arr,i,i_2,j_2,1))+1+4)$ (programa Array)

$= ACCESS(GRD(arr,i),(i_2,j_2))$ (prog. G8)

$= ACCESS(gr,(i_2,j_2))$ (*4) (por definição)

De (*1), (*2), (*3) e (*4) vem

ACCESSG(ASSIGNG(gr,(i_1,j_1),r),(i_2,j_2))=

IF $i_1=i_2$ Λ $j_1=j_2$

THEN IF ISING(gr,(i_1,j_1))

THEN r (*1)

ELSE UNDEFINED (*2)

ELSE ACCESSG(gr,(i_2,j_2)) (*3) e (*4)

ISINDCL(DECLARENEWGRID(rtgr),(i_1,j_1))=

=ISINDCL(DCL(NEWLIST(RTG(ABSCISSAINFR(rtgr),ABSCISSASUPR(rtgr),ORDENADAINFR(rtgr),ORDENADASUPR(rtgr)))),(i_1,j_1)) (progr. G1)

=ISINL(NEWLIST(RTG(ABSCISSAINFR(rtgr), ABSCISSASUPR(rtgr),ORDENADAINFR(rtgr), ORDENADASUPR(rtgr))),(i_1,j_1)) (progr. G4)

=ISIN(RTG(ABSCISSAINFR(rtgr),ABSCISSASUPR(rtgr),ORDENADAINFR(rtgr),ORDENADASUPR(rtgr), (i_1,j_1))) (pela definição de (ISINL e porque ISRETANG(RTG(ABSCISSAINFR(rtgr),ABSCISSASUPR(rtgr),ORDENADAINFR(rtgr),ABSCISSASUPR(rtgr)))=TRUE)

=ABSCISSAINFR(rtgr) $\leq i_1 \leq$ ABSCISSASUPR(rtgr) Λ ORDENADAINFR(rtgr) $\leq j_1 \leq$ ORDENADASUPR(rtgr) (programa RT12)

=ISINR(rtgr,(i_1,j_1)) (por definição ISINR em Retanggr)

ISINDCL(DECLAREGPLUS(dcl,rtgr),(i_1,j_1))=

=ISINDCL(DECLAREGPLUS(DCL(retlist),(rtgr),(i_1,j_1))) (por definição)

=ISINDCL(DCL(INSERT(retlist,RTG(ABSCISSAINFR(rtgr), ABSCISSASUPR(rtgr),ORDENADAINFR(rtgr), ORDENADASUPR(rtgr)))),(i_1,j_1))) (progr. G2)

=ISINL(INSERT(retlist,RTG(ABSCISSAINFR(rtgr), ABSCISSASUPR (rtgr),ORDENADAINFR(rtgr), ORDENADASUPR (rtgr))),(i_1,j_1))) (progr. G4)

=ISINL(retlist,(i_1,j_1))) V ISIN(RTG(ABSCISSAINFR(rtgr), ABSCISSASUPR(rtgr),ORDENADAINFR(rtgr), ORDENADASUPR (rtgr))),(i_1,j_1))) (pela definição ISINL e ISRETANG (RTG(ABSCISSAINFR(rtgr),ABSCISSASUPR(rtgr), ORDENADAINFR(rtgr),ORDENADASUPR(rtgr))=TRUE)

= ISINL(retlist,(i_1,j_1))) V (ABSCISSAINFR(rtgr) $\leq i_1 \leq$ ABSCISSASUPR(rtgr) $\wedge$ ORDENADAINFR(rtgr) $\leq j_1 \leq$ ORDENADASUPR(rtgr) (progr. RT12)

=ISINDCL(DCL(retlist),(i_1,j_1))) V ISINR(rtgr,(i_1,j_1))) (programa G4 e definição de ISINR em Retanggr)

=ISINDCL(dcl,(i_1,j_1))) V ISINR(rtgr,(i_1,j_1))) (por def.)

ISINDCL(DECLAREGMINUS(dcl(rtgr),(i_1,j_1))=

=ISINDCL(DECLAREGMINUS(DCL(retlist),(rtgr),(i_1,j_1))) (por definição)

=ISINDCL(DCL(DELETE(retlist,RTG(ABSCISSAINFR(rtgr), ABSCISSASUPR(rtgr),ORDENADAINFR(rtgr), ORDENADASUPR(rtgr)))),(i_1,j_1))). (programa G3)

=ISINL(DELETE(retlist,RTG(ABSCISSAINFR(rtgr), ABSCISSASUPR
(rtgr),ORDENADAINFR(rtgr),
ORDENADASUPR(rtgr))),(i_1,j_1))
(prog. G4)

=ISINL(retlist,(i_1,j_1)) $\wedge \neg$ ISIN(RTG(ABSCISSAINFR(rtgr),
ABSCISSASUPR(rtgr),ORDENADAINFR(rtgr),
ORDENADASUPR(rtgr)),(i_1,j_1)) (pela definição
de ISINL e porque ISRETANG(RTG(ABSCISSAINFR
(rtgr),ABSCISSASUPR(rtgr),ORDENADAINFR
(rtgr),ORDENADASUPR(rtgr))=TRUE)

=ISINL(retlist,(i_1,j_1)) $\wedge \neg$ (ABSCISSAINFR(rtgr) $\leq i_1 \leq$
ABSCISSASUPR(rtgr) $\wedge$ ORDENADAINFR(rtgr) $\leq$
$j_1 \leq$ ORDENADASUPR(rtgr)) (Prog. RT12)

=ISINDCL(DCL(retlist),(i_1,j_1)) $\wedge \neg$ (ISINR(rtgr,(i_1,j_1))
(programa G4 e definição de ISINR em Retanggr)

=ISINDCL(dcl,(i_1,j_1)) $\wedge \neg$ ISINR(rtgr,(i_1,j_1)) (por def.)

ISING(ASSIGNG(dcl,(i_1,j_1),r),(i_2,j_2))=

=ISING(ASSIGNG(DCL(retlist),(i_1,j_1),r),(i_2,j_2))=

=ISING(GRD(arr,i),(i_2,j_2)) onde

GRD(arr,i)=ASSIGNG(DCL(retlist),(i_1,j_1),r)

onde arr = RESERVM(DOLIST(retlist) e

i = SIZEARR(DOLIST(retlist))

Então se SCANG(arr,i,i_2,j_2,1)=UNDEFINED tem-se

ISING(GRD(arr,i),(i_2,j_2))=FALSE (por G6), ISINL(retlist, i_1,
j_2)) = FALSE (por A5) e

ISINDCL(DCL(retlist),(i_2,j_2))= FALSE (por G4)

Portanto

ISING(ASSIGNG(DCL(retlist),(i_1,j_1),r),(i_2,j_2))=

=ISINDCL(DCL(retlist), (i_2,j_2))

Por outro lado, se SCANG(arr,i,i_2,j_2,1) ≠ UNDEFINED tem-se

ISING(GRD(arr,i),(i_2,j_2)) = FALSE (por G6),

ISINL(retlist,(i_2,j_2))= FALSE

e ISINDCL(DCL(retlist),(i_2,j_2))=FALSE (por G4).

Logo,

ISING(ASSIGNG(dcl,(i_1,j_1),r),(i_2,j_2))=ISINDCL(dcl,(i_2,j_2))

ISING(ASSIGNG(gr,(i_1,j_1),r),(i_2,j_2))=

=ISING(ASSIGNG(GRD(arr,i),(i_1,j_1),r),(i_2,j_2))=

Se SCANG(arr,i,i_1,j_1,1)=UNDEFINED então

ISING(ASSIGNG(GRD(arr,i),(i_1,j_1),r),(i_2,j_2))=

=ISING(GRD(arr,i),(i_2,j_2)) (*1) (prog. G7)

Se SCANG(arr,i,i_1,j_1,1) ≠ UNDEFINED então

ISING(ASSIGNG(GRD(arr,i),(i_1,j_1),r),(i_2,j_2))=

=ISING(GRD(arr_1,i),(i_2,j_2)) onde

arr_1=ASSIGN(arr,SCANG(arr,i,i_1,j_1,1)+(j_1-ACCESS(arr, SCANG (arr,i,i_1,j_1,1)+2+1)*(i_1-ACCESS(arr,SCANG(arr, i,i_1,j_1,1))+1)+(i_1-ACCESS(arr,SCANG(arr,i,i_1,j_1, 1))+1+4,r)

note que

SCANG(arr_1,i,i_2,j_2,1)=SCANG(arr,i,i_2,j_2,1)

já que na passagem de arr para arr_1 as posições dos elementos não foram alteradas.

Então se SCANG(arr_1,i,i_2,j_2,1)=UNDEFINED tem-se

SCANG(arr,i,i_2,j_2,1) = UNDEFINED

isto é,

ISING(GRD(arr_1,1),(i_2,j_2))=FALSE → ISING(GRD(arr,i), (i_2, j_2))= FALSE (programa G6) (*1)

e se ISING(GRD(arr_1,1),(i_2,j_2))=TRUE → ISING(GRD(arr,i) (i_2, j_2)) = TRUE (programa G6) (*2)

Por (*1) e (*2) vem que

ISING(GRD(arr_1,i),(i_2,j_2))=ISING(GRD(arr,i),(i_2,j_2))

ou seja,

ISING(ASSIGNG(gr,(i_1,j_1),r),(i_2,j_2))=ISING(gr, (i_2,j_2))

Demonstração das Funções Auxiliares

(1) RTLISTTOLIST(Retanglist, $Integer^+$) → List

Seja retlist uma Retanglist e i um $Integer^+$

RTLISTTOLIST(retlist,i)=

= IF i = 1

THEN INLIST(CREATE,1,RETURN(retlist,1))

ELSE INLIST(RTLISTTOLIST(retlist,i-1),i,RETURN (retlist,i))

Esta função forma uma List, a partir de retlist, inserindo um a um os i+1 elementos de retlist na List que é inicialmente vazia.

Se i = 1 então

RTLISTTOLIST(retlist,1)=INLIST(CREATE,1,RETURN(retlist,1))

Sabe-se que a operação RETURN(retlist,1) tem como resultado um Retang que é o primeiro elemento de retlist e que a operação INLIST insere um Retang em uma List. Neste caso tem-se a inserção do Retang, obtido através da operação RETURN, na primeira posição de uma List vazia, obtida pela operação CREATE.

Considerar que para $1 \leq i \leq n$, $n \in N$ a operação RTLISTTOLIST(retlist,i) cria uma List e insere nela os i primeiros Retang's de retlist.

Então, para i = n + 1 tem-se

RTLISTTOLIST(retlist,i + 1)=INLIST(RTLISTTOLIST(retlist,i), i + 1, RETURN(retlist,i + 1))

A operação RETURN(retlist, i + 1) tem como resultado o (i + 1)-ésimo Retang de retlist e por hipótese, RTLISTTOLIST(retlist,i) insere os i primeiros elementos de retlist em uma List. Então, INLIST(RTLISTTOLIST(retlist,i), i + 1, RETURN(retlist, i + 1)) insere o (i + 1)-ésimo Retang de retlist, na (i + 1)-ésima posição de List formada por RTLISTTOLIST(retlist,i).

Logo RTLISTTOLIST(retlist, i + 1) forma uma List, a partir da Retanglist, retlist, inserindo um a um os i + 1 elementos de retlist na List que é inicialmente vazia.

(2) DELINTFIRST(Retanglist, List, $Integer^+$, $Integer^+$) → (Retanglist, List, $Integer^+$, $Integer^+$)

Sejam retlist uma Retanglist, lst uma List e j, k Integer^{+}'s

DELINTFIRST(retlist,lst,k,j)=(retlist,JOIN(lst,RTLISTTOLIST
(RETURN(retlist,j)-(RETURN(retlist,j) ∩ RETURN
(retlist,1)),4),k),k+SIZEL(RTLISTTOLIST(RETURN
(retlist,j)-(RETURN(retlist,j)∩ RETURN(retlist,1)),
4), j-1)

A operação RETURN(retlist,j)-(RETURN(retlist,j) ∩ RETURN(retlist,1)) dá origem a uma Retanglist que é formada pela retirada da secção do j-ésimo Retang de retlist que corresponde a sua intersecção com o primeiro Retang de retlist. RTLISTTOLIST irá criar uma List com todos os Retang's que compõem a Retanglist.

A operação SIZEL(List) devolve o número de elementos da List, então SIZEL(RTLISTTOLIST(RETURN(retlist, j) - (RETURN(retlist,j) ∩ RETURN(retlist,1)),4) terá como resultado o número de elementos da List formada pela operação RTLISTTOLIST.

A operação JOIN(lst,RTLISTTOLIST(RETURN(retlist,
j) - (RETURN(retlist,j) ∩ (RETURN
(retlist,1)),4),k) insere na posição k + 1 de lst a List fornecida por RTLISTTOLIST.

Então a função

DELINTOFIRST(retlist,lst,k,j)

insere na posição k+1 de lst a List resultante da operação de diferença entre o j-ésimo e o primeiro Retang da retlist. O endereço para a próxima inserção (k) fica alterado para k + tamanho da List inserida. O próximo Retang a ser recuperado para a

comparação com o primeiro é aquele que ocupa a posição j - 1 de retlist.

(3) DELINTOTHERS(List, Integer$^+$, Integer$^+$) → List

Seja lst uma List e i, j Integer$^+$'s

DELINTOTHERS(lst,j,i)=

IF i ≤ 0 THEN lst

ELSE DELIST(JOIN(DELINTOTHERS(lst,j,i-1),
RTLISTTOLIST(ELEMENT(DELINTOTHERS
(lst,j,i-1),i+j)-(ELEMENT(
DELINTOTHERS(lst,j,i-1),i+j)∩ ELEMENT
(DELINTOTHERS(lst,j,i-1),j)), i+j)i +
j)

Se i = 0 então
DELINTOTHERS(lst,j,i) = lst
Para i = 1 tem-se
DELINTOTHERS(lst,j,1)=DELIST(JOIN(DELINTOTHERS(lst,j,0),
RTLISTTOLIST(ELEMENT(DELINTOTHERS(lst, j,
0),j+1)-(ELEMENT(DELINTOTHERS(lst,j,0),j+
1)∩ELEMENT(DELINTOTHERS(lst,j,0),4),j+1),j
+1)=DELIST(JOIN(lst,RTLISTTOLIST(ELEMENT
(lst,j+1) - (ELEMENT(lst,j+1) ∩ (ELEMENT
(lst,j),4),j+1),j+1)

onde ELEMENT(lst,j) devolve o j-ésimo Retang de lst e ELEMENT (lst,j+1)-(ELEMENT(lst,j+1) ∩ ELEMENT(lst,j)) devolve uma Retanglist formada pela diferença entre o (j+1)-ésimo Retang de lst e sua intersecção com o j-ésimo Retang de lst. Esta Retang-

list será transformada em uma List pela função RTLISTTOLIST.

A operação JOIN insere a List fornecida por RTLISTTOLIST após a posição j+1 da List. (DELINTOTHERS(lst, j, 0) e a operação DELIST retira o (j+1)-ésimo elemento da List resultante do JOIN. Note que o (j+1)-ésimo elemento da List resultante é o mesmo que (j+1)-ésimo elemento de lst uma vez que a inserção é efetuada após a posição j+1.

Então, para i = 1, a função DELINTOTHERS retira o (j+1)-ésimo elemento de lst, inserindo nesta posição, uma List obtida a partir de uma Retanglist que é o resultado da diferença entre o (j+1)-ésimo elemento e sua intersecção com o j-ésimo.

Supor que para $i \leq i \leq n$, $n \in N$, a função DELINTOTHERS elimina de lst o (j+1)-ésimo elemento de lst e insere em seu lugar uma List obtida a partir de uma Retanglist que é o resultado da diferença entre o (j+1)-ésimo elemento e sua intersecção com o j-ésimo, para i = 1,...,n.

Então, para i = n+1, tem-se

DELINTOTHERS(lst,j,i)=DELIST(JOIN(DELINTOTHERS(lst,j,i-1),
RTLISTTOLIST(ELEMENT(DELINTOTHERS(lst,j,i-1),j+i)-(ELEMENT(DELINTOTHERS(lst,j, i - 1), (j+i) ∩ ELEMENT(DELINTOTHERS(lst,j,i-1),j),4), j+i), j+i)

Por hipótese de indução DELINTOTHERS(lst,j,i-1) retira de cada elemento da lst desde o (j+1)-ésimo até o (j+i-1)-ésimo sua intersecção com o j-ésimo elemento e devolve uma List que satisfaz o que se espera da função. Note que a operação

JOIN acrescenta a esta List exatamente a diferença do (j+1)-ésimo elemento de List com sua intersecção com o j-ésimo elemento o que prova que a função realmente faz o que se espera dela.

(4) DOMINUS(List, Integer^{+}, Integer^{+}) → List

Seja lst uma List e i, j Integer^{+}'s com i < j e j > i

Esta é uma função parcial, só está definida por i < j

```
DOMINUS(lst,j,i)=
      = IF i ≤ 1
           THEN lst
           ELSE DELIST(JOIN(DOMINUS(lst,j,i-1),RTLISTTOLIST
                (ELEMENT(DOMINUS(lst,j,i-1),i-1)-(ELEMENT(
                DOMINUS(lst,j,i-1),i-1)  ELEMENT(DOMINUS(lst,
                j,i-1),j)),i-1),i-1)
```

Se i ≤ 1 então por definição

DOMINUS(lst,j,i)=lst

Para i = 2 tem-se

```
DOMINUS(lst,j,2)=DELIST(JOIN(DOMINUS(lst,j,1),RTLISTTOLIST
                (ELEMENT(DOMINUS(lst,j,1),1)-(ELEMENT(DOMINUS
                (lst,j,1),i-1)∩ ELEMENT(DOMINUS(lst,j,1),j)),
                4),1),1)=DELIST(JOIN(lst,RTLISTTOLIST(ELEMENT
                (lst,1)-(ELEMENT(lst,1)∩ ELEMENT(lst,j)),4),
                1),1)
```

A operação ELEMENT(lst,1)-(ELEMENT(lst,1) ∩ ELEMENT (lst,j)) resulta em uma Retanglist formada pela diferença entre o primeiro elemento de lst e sua intersecção com o j-ésimo ele-

mento. Esta Retanglist é transformada em uma List pela função RTLISTTOLIST. A operação JOIN junta as duas List's inserindo os elementos da List obtida pela função RTLISTTOLIST após a primeira posição de lst. DELIST retira o primeiro elemento de lst.

Suponha que para n ∈ N, a função DOMINUS retire o (i-1)-ésimo elemento de lst, inserindo a partir da posição i uma List obtida pela aplicação da função RTLISTTOLIST sobre a Retanglist formada pela diferença entre o i-ésimo elemento de lst e sua intersecção com o j-ésimo elemento de lst. Note que o processo é repetido para i = 2, 3,...n, isto é, i-1 indica o número de elementos que devem ser eliminados de lst e em cujas posições são inseridas List's formadas pela aplicação de RTLISTTOLIST sobre Retanglist's obtidas de maneira descrita anteriormente.

Então, para i = n+1 tem-se:

```
DOMINUS(lst,j,i)=DELIST(JOIN(DOMINUS(lst,j,i-1),RTLISTTOLIST
                (ELEMENT(DOMINUS(lst,j,i-1),i-1)-(ELEMENT
                (DOMINUS(lst,j,i-1),i-1) ∩ ELEMENT(DOMINUS
                (lst,j,i-1),j)),4),i-1),i-1)
```

A função DOMINUS(lst,j,i-1) devolve uma List onde foram inseridas, nas posições de 2 até i-1 as List's formadas pela aplicação de RTLISTTOLIST sobre cada uma das Retanglist's obtidas da operação de diferença entre o k-ésimo elemento (k = 1, 2...,i-2) e sua intersecção com o j-ésimo elemento.

O resultado final da operação JOIN é a inserção à List resultante de DOMINUS(lst,j,i-1) a List formada pela aplicação de RTLISTTOLIST sobre a Retanglist obtida através da operação de diferença entre (i-1)-ésimo elemento e sua intersecção

com o j-ésimo elemento, no lugar do (i-1)-ésimo elemento que é retirado por DELIST.

Portanto, a função DOMINUS(lst,j,i) faz o que se espera dela.

(5) SCANLIST(List, Integer^{+}) → List

Seja lst uma List e j um Integer^{+}

SCANLIST é uma função parcial, não está definida para lst se SIZEL(lst) ≤ 4.

```
SCANLIST(lst,j)=
        = IF j ≤ 1
             THEN lst
             ELSE SCANLIST(DELINTOTHERS(lst,j-1,SIZEL(lst)
                          -j-2),j-1)
```

Se j ≤ 1, então, por definição,
SCANLIST(lst,j)=lst
Se j = 2, tem-se

```
SCANLIST(lst,2)=SCANLIST(DELINTOTHERS(lst,1,SIZEL(lst)-2-2),1)=
                DELINTOTHERS(lst,1,SIZEL(lst)-4)      (por def.)
```

caso SIZE(lst) ≤ 4 tem-se que
SCANLIST(lst,2) não está definida.

Caso contrário, DELINTOTHERS(lst,1,SIZEL(lst)-4) irá percorrer lst a partir da segunda posição até a posição SIZE(lst)-3, fazendo a diferença entre cada elemento e sua intersecção com o primeiro, (isto é, o (j-1)-ésimo elemento) obtendo, assim uma

Retanglist que será transformada em uma List de acordo com a definição de DELINTOTHERS.

Supor que para $2 \leq j \leq n$, $n \in N$, a função DELINTOTHERS seja aplicada sobre os elementos desde a primeira posição em lst até a posição n-1.

Então, para j = n+1, tem-se

SCANLIST(lst,n+1)=SCANLIST(DELINTOTHERS(lst,n,SIZEL(lst)- n-1), n) onde DELINTOTHERS é aplicada sobre o elemento que ocupa a n-ésima posição em lst e por hipótese de indução DELINTOTHERS é aplicada aos elementos desde o 2º até a (n-1)-ésima posição.

Portanto, a função SCANLIST serve para percorrer lst aplicando a função DELINTOTHERS sobre cada elemento de lst situado desde a segunda posição até a posição SIZEL(lst)-j-2.

(6) SCANRETLIST(Retanglist, List, Integer$^+$, Integer$^+$) →
(Retanglist, List, Integer$^+$, Integer$^+$)

Sejam retlist uma Retanglist, lst uma List, i, j Integer$^+$'s

```
SCANRETLIST(retlist,lst,k,j)=
        = IF RETSIG(retlist,j) = +
             THEN DELINFIRST(retlist,lst,k,j)
             ELSE (retlist,DOMINUS(lst,j,SIZEL(lst),k,j-1)
```

A função RETSIG(retlist,j) examina o sinal do j-ésimo elemento de retlist. Caso RETSIG(retlist,j) = + a função DELINFIRST é aplicada sobre o j-ésimo elemento de retlist, isto é, insere em lst o j-ésimo elemento de retlist menos sua intersecção com todos os elementos que o precederam em lst. Caso con

trário, a função DOMINUS é aplicada para todos os elementos lst, isto é, é retirada de cada elemento de lst sua intersecção com o j-ésimo elemento de retlist. Portanto, SCANRETLIST (retlist,lst,k,j) transmitirá a lst o efeito do j-ésimo elemento de retlist.

(7) FIRSTVERLIST(Retanglist, Integer) → List

Seja retlist uma Retanglist e j um Integer

```
FIRSTVERLIST(retlist,j)=
        = IF j = 1
            THEN INLIST(CREATE,0,RETURN(retlist,1))
            ELSE SCANRETLIST(retlist,FIRSTVERLIST(retlist,
                                j-1),1,j)
```

Se j = 1 então, por definição,

FIRSTVERLIST(retlist,1)=INLIST(CREATE,0,RETURN(retlist,1))

que cria uma List e insere na primeira posição desta List o Retang obtido pela operação RETURN(retlist,1). Lembre que o primeiro elemento de Retanglist é sempre um Retang positivo, portanto, o efeito de FIRSTVERLIST(retlist,1) é criar uma List e inserir nela um Retang positivo.

Supor que para $n \in N$, FIRSTVERLIST(retlist,n) cria uma List e insere nela os Retang's que formam a região definida pelos n primeiros elementos de retlist.

Seja j = n+1

```
FIRSTVERLIST(retlist,j)=SCANRETLIST(retlist,FIRSTVERLIST
                                (retlist,n),1,n+1)
```

Por hipótese de indução FIRSTVERLIST(retlist,n) é uma List com os Retang's que formam a região definida pelos n primeiros elementos de retlist. Através de SCANRETLIST é inserida a esta List o (n+1)-ésimo elemento de retlist menos sua intersecção com os outros elementos de List se este elemento é um Retang positivo e o (n+1)-ésimo elemento de retlist é retirado dos elementos da List em questão.

Portanto, para qualquer efeito de FIRSTVERLIST (retlist,j) é criar uma List com os Retang's que formam a região definida pelos n primeiros elementos de retlist.

(8) DOLIST(Retanglist) → List

Seja retlist uma Retanglist

Então, DOLIST(retlist)=SCANLIST(FIRSTVERLIST(retlist,SIZE
(retlist)),SIZEL(FIRSTVERLIST
(retlist,SIZE(retlist))))

A função FIRSTVERLIST percorre a retlist, criando a primeira versão da List do modo descrito anteriormente.

A função SCANLIST percorre a List obtida a partir da função FIRSTVERLIST retirando as intersecções ainda existentes entre os Retang's.

Portanto, DOLIST fornece a partir de uma Retanglist, uma List de Retang's disjuntos que formam a região definida pela Retanglist.

(9) SCANG(Array, $Integer^+$, Integer, Integer, $Integer^+ - \{0\}$) → $Integer^+ - \{0\}$

Seja arr um Array, i_1, j_1 dois Integer's

e i,k dois $Integer^+$'s $k \neq 0$

SCANG(arr,i,i_1,j_1,k) = IF $i \geq k + 4$

THEN IF ACCESS(arr,k)$\leq i_1 \leq$ACCESS(arr, k+1), $\wedge$

ACCESS(arr,k+2)$\leq j_1 \leq$ACCESS (arr, k+3)

THEN k

ELSE SCANG(arr,i,i_1,j_1,k+ ACCESS(arr,k+4)+5)

ELSE UNDEFINED

A função SCANG foi definida com o objetivo de procurar o ponto (i_1, j_1) no Array arr, percorrendo-o a partir de posição k até a posição i-4 testando as coordenadas dos Retang's ali armazenados. Devolve o endereço do cabeçalho do Retang que contém o ponto. Se o ponto (i_1, j_1) não estiver entre as posições k e i-4 o resultado de função é UNDEFINED.

SCANG percorre o Array arr desde a posição k até a posição i-4 testando o conteúdo das posições k e k+1 com i_1 e das posições k+2 e k+3 com i_2. Caso ACCESS(arr,k)$\leq i_1 \leq$ ACCESS (arr,k+1)$\wedge$ ACCESS(arr,k+2)$\leq j_1 \leq$ACCESS(arr,k+3) então o processo termina e o resultado de SCANG é o valor k, caso contrário, o mesmo teste será repetido para k valendo k+ACCESS(arr,k+4) +5. Caso o novo valor de k seja maior que i-4 então o processo pára e o resultado é uma mensagem de indefinido, indicando que o pon

to (i_1,j_1) não foi encontrado.

Então o processo recursivo pode ser interrompido por dois motivos: uma condição é satisfeita ou o valor i-4 é ultrapassado.

Seja K(arr) = {k/k é o endereço de um cabeçalho de arr} um conjunto ordenado de inteiros.

Base de indução:

Seja k=max K(arr)=i-5 então o ponto (i_1,j_1) não está entre as posições k e i-4 portanto SCANG(arr,i,i_1,j_1,k)=UNDEFINED

Suponha que $\forall$ n > k, n$\in$ K(arr), SCANG(arr,i,i_1,j_1,n) = posição do ponto (i_1,j_1) em arr se este ponto estiver entre as posições n e i-4 ou UNDEFINED caso contrário.

Seja n o elemento logo anterior a k em K(arr).

Se ACCESS(arr,n)$\leq i_1 \leq$ACCESS(arr,n+1) e

ACCESS(arr,n+2)$\leq j_1 \leq$ACCESS(arr,n+3)

então o ponto está no Retang definido por este cabeçalho e o resultado é n. Senão SCANG é chamada com o quarto elemento de quádrupla valendo k+ACCESS(arr,k+4)+5 que é o próximo elemento de K(arr) e então se aplica a hipótese de indução.

(10) RESERVM(List) → Array

Seja lst uma List

RESERVM(lst) = π3 (RM(CREATEARG(lst)))

π3 (RM(lst,1,NEWARRAY,SIZE(lst))

A função RM devolve uma quádrupla onde o terceiro elemento é um Array formado a partir de lst. A função π_3 recebe uma quádrupla e devolve o terceiro elemento da quádrupla. Então π_3 (RM(lst,1,NEWARRAY,SIZEL(lst))) tem como resultado um array que contém as informações sobre os elementos de lst mais as posições reservadas para os pontos de cada Retang de lst.

Logo, RESERVM(lst) fornece, a partir de lst, um Array que irá conter, para cada Retang de lst, o cabeçalho do Retang e as posições reservadas para os pontos do Retang.

(11) π_2(List, Integer$^+$,Array,Integer$^+$) → Integer$^+$

Seja lst uma List, arr um Array e k,j dois Integer$^+$'s

π_2(lst,k,arr,j)=k

Esta função leva um quádrupla no segundo elemento da quádrupla.

(12) π_3(List, Integer$^+$,Array,Integer$^+$) → Array

Seja lst uma List, arr um Array e k, j dois Integer$^+$'s

π_3(lst,k,arr,j)= arr

Esta função leva uma quádrupla no terceiro elemento da quádrupla.

(13) CREATEARG(List) → (List,1,Array,Integer$^+$)

Seja lst uma List

CREATEARG(lst)=(lst,1,NEWARRAY,SIZEL(lst))

Esta função recebe uma List e fornece, como resultado, a mesma List, o valor 1, um Array vazio e o tamanho da List recebida, pois a operação NEWARRAY cria um Array vazio e a operação SIZEL(lst) devolve o número de elementos de lst.

(14) SIZEARR(List) → $Integer^{+}$

Esta função recebe uma List, e devolve o tamanho do array resultante da reserva de espaço para os pontos definidos pelos Retang's da List.

Seja lst uma List

SIZEARR(lst) = π_2(RM(CREATEARG(lst))

= π_2(RM(lst,1,NEWARRAY,SIZEL(lst))

A função π_2 recebe uma quádrupla e devolve o segundo elemento desta quádrupla. A função RM devolve uma quádrupla. Conforme definição de RM o segundo elemento desta quádrupla é o número total de posições reservadas no array para conter os pontos de todos os Retang's pertencentes à lst mais os cabeçalhos para cada Retang de lst.

Então SIZEARR(lst) tem como resultado um valor inteiro que indica o número de posições ocupadas no array formado pela função RM a partir de lst.

(15) RM(List, Integer,Array,$Integer^{+}$) → (List,Integer,Array, Integer).

Seja lst uma List, arr um Array e j, k dois $Integer^{+}$'s então

```
RM(lst,k,arr,j)=IF  j > 0
                    THEN RM(lst,k+5+(ABSCISSASUP(ELEMENT(lst,
                         j))-ABSCISSAINF (ELEMENT(lst,j)))+1)*
                         ORDENADASUP (ELEMENT(lst,j)) -
                         ORDENADAINF (ELEMENT(lst,j))+1),
                         ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN
                         (arr,k,ABSCISSAINF (ELEMENT(lst,j))),
                         k+1,ABSCISSASUP(ELEMENT(lst,j))),
                         k+2,ORDENADAINF(ELEMENT(lst,j))),
                         k+3,ORDENADASUP(ELEMENT(lst,j))),
                         k+4,(ABSCISSASUP(ELEMENT(lst,j))-
                         ABSCISSAINF(ELEMENT(lst,j))+1)*
                         ORDENADASUP(ELEMENT(lst,j))-
                         ORDENADAINF(ELEMENT(lst,j))+1),j-1)
```

A operação ELEMENT(lst,j) fornece, como resultado, o j-ésimo elemento de lst. Seja rtg este elemento onde rtg é um Retang composto pelos inteiros i_1, i_2, i_3, i_4 nesta ordem. Então, as operações ABSCISSAINF, ABSCISSASUP, ORDENADAINF e ORDENADASUP sobre rtg fornecem quatro inteiros i_1, i_2, i_3, i_4, respectivamente. Estes quatro inteiros, mais o valor $(i_2-i_1+1)*(i_4-i_3+1)$ que é o número de inteiros do Retang rtg formam o cabeçalho da área que será reservada para os elementos desse Retang e são inseridos em arr nas posições k, k+1, k+2 e k+3 respectivamente. k é então a primeira posição livre de arr e tem portanto que ser atualizado.

Cada aplicação de RM para j > 0 altera o valor de k para $k+5+(i_2-i_1+1)*(i_4-i_3+1)$ e o valor de j para j-1. Então, enquanto j decresce, k aumenta. Quando j = 0 a função atribui o

valor 0(zero) na posição k+4 de arr e altera o valor de k para k+4.

Então, RM recupera cada elemento de lst desde o j-ésimo até o primeiro inserindo em arr os inteiros que identificam cada um dos j-elementos de lst, seguidos de um valor calculado a partir destes inteiros conforme mostrou-se anteriormente.

Após a inserção dos j elementos de lst em arr, quatro posições de arr são saltadas e na quinta é colocado o valor zero.

Supor que para j = n, n ∈ N, a função de RM seja aplicada n+1 vezes, inserindo em arr as informações obtidas dos n elementos de lst, da forma descrita anteriormente.

Então, para j = n+1 tem-se

```
RM(lst,k,arr,n+1)=RM(lst,k+5(ABSCISSASUP(ELEMENT(lst,n+1))-
                  ABSCISSAINF(ELEMENT(lst,n+1))+1)*
                  ORDENADASUP(ELEMENT(lst,n+1))-
                  ORDENADAINF(ELEMENT(lst,n+1))+1),ASSIGN
                  (ASSIGN(ASSIGN(ASSIGN(ASSIGN(arr,
                  k,ABSCISSAINF(ELEMENT(lst,n+1)),
                  k+1,ABSCISSASUP(ELEMENT(lst,n+1)),
                  k+2,ORDENADAINF(ELEMENT(lst,n+1)),
                  k+3,ORDENADASUP(ELEMENT(lst,n+1)),
                  k+4,(ABSCISSASUP(ELEMENT(lst,n+1))-
                  ABSCISSAINF(ELEMENT(lst,n+1))+1)*(ORDENADASUP
                  (ELEMENT(lst,n+1))-ORDENADAINF(ELEMENT(lst,n+
                  1))+1),n)
```

Pode-se ver que o cabeçalho correspondente ao j-ésimo elemento é inserido em arr e k(segundo elemento da quádrupla) será atualizado com a primeira posição livre de arr.

APÊNDICE 2 - CORREÇÃO DA LIST

CORREÇÃO DA LIST

Suponha que existe

- Um Array arr e um inteiro não negativo i tal que

  list = LST(arr,i)

- os inteiros não negativos j, k
- um Retang rtg_1
- uma List $list_1$ e um array arr_1

DELIST(CREATE, i) = DELIST(LST(NEWARRAY,0),j) (por def.)

= LST(NEWARRAY,0) (prog . 3)

= CREATE (por def.)

JOIN(list,CREATE,j) = JOIN(LST(arr,i),LST(NEWARRAY,0),j) (por def.)

= LST(arr,i) (prog. 10)

= list (por def.)

SIZEL(CREATE) = SIZEL(LST(NEWARRAY,0)) (por def.)

= DOSIZE(LST(NEWARRAY,0),0) (prog. L4)

= 0 (prog. L11 e L21)

SIZEL(INLIST(list,j, elm))=

= SIZEL(INLIST(LST(arr,i),j,rtg_1) (por def.)

Suponha j > SIZEL(LST(arr,i)) então

SIZEL(INLIST(list,j,elm))=SIZEL(LST(arr.i)) (prog. L2)

= SIZEL(list) (por def.)

Por outro lado, se i=SIZEL(LST(arr,i)) tem-se

SIZEL(INLIST(LST(arr,i),j.rtg_1)=

=SIZEL(LST(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(arr,i+5,

i+6), i, i+1),

i+1, ABSCISSAINF(rtg_1)).

i+2, ABSCISSASUP(rtg_1)),

i+3, ORDENADAINF(rtg_1)).

i+4, ORDENADASUP(rtg_1)), i+5) (prog. L2)

= DOSIZE(LST(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(arr,i+5,

ACCESS(arr,NEXT(arr,i,j-1)+4)),NEXT(arr,i,j-1)+4,i+1),

i+1,ABSCISSAINF(rtg_1)).

i+2,ABSCISSASUP(rtg_1)),

i+3,ORDENADAINF(rtg_1)),

i+4,ORDENADASUP(rtg_1)),i+5),0)

Observe que o programa DOSIZE percorre a lista desde o primeiro até o enésimo elemento com a utilização do programa NEXT que fornece o elemento seguinte a um determinado elemento.

A iteração termina quando o programa NEXT devolve o endereço do próximo elemento como sendo o tamanho do Array acrescido de uma unidade.

Para observar o funcionamento do DOSIZE suponha, em primeiro lugar, que rtg_1 seja inserido no final da lista, isto é, j = SIZEL(list).

Então, na posição i+5 será atribuído o valor i+6 e na posição i, o valor i+1.

Desta forma, o programa DOSIZE irá percorrer o array resultante do INLIST da mesma maneira como percorreria o array

arr até a posição i com a diferença de que no array resultante haverá mais uma iteração (mais um NEXT) para acessar o último elemento inserido. Então o valor de SIZEL(INLIST(list,j,elm)) será igual ao SIZEL(list) + 1.

Da mesma forma, se rtg_1 é inserido em uma posição intermediária de list, na posição i+5 será atribuído o endereço do elemento que segue o j-ésimo elemento ACCESS(arr,NEXT(arr, i, j-1)+4)) e na posição NEXT(arr,i,j-1)+4 será atribuído o valor i+1. O último elemento de list não será alterado e continuará sendo o último elemento da nova lista.

Assim, o programa DOSIZE percorrerá a List resultante da mesma forma que a List original até encontrar o elemento inserido. Neste ponto será efetuado um NEXT a mais que fornecerá o endereço do j+1-ésimo elemento da lista original que passa a ser, depois da inserção, o j+2-ésimo elemento da List resultante, isto é, todos os elementos que seguem o elemento inserido ocuparão uma posição, na List resultante, uma unidade acima daquela que ocupavam na List original. Isto significa que o programa DOSIZE terminará de percorrer a List resultante quando encontrar o último elemento da List original, que também será o último elemento da List resultante, e que ocupará a posição SIZEL(list)+1.

Portanto, SIZEL(INLIST(list,j,rtg_1))=

```
IF j ≤ SIZEL(list)
   THEN SIZEL(list) + 1
   ELSE SIZEL(list)
```

SIZEL(DELIST(list,j)) =

= SIZEL(DELIST(LST(arr,i),j)) (por definição)

Suponha j = 0 então

SIZEL(DELIST(list,0))=SIZEL(DELIST(LST(arr,i),0))

= SIZEL(LST(arr,i)) (prog. L3)

= SIZEL(list) (por definição)

Por outro lado, se j > SIZEL(list) tem-se

SIZEL(DELIST(list,j))=SIZEL(DELIST(LST(arr,i),j))

= SIZEL(LST(arr,i)) (prog. L3)

= SIZEL(list) (por definição)

Suponha, agora j $\leq$ SIZEL(list), então

SIZEL(DELIST(list,j))=SIZEL(DELIST(LST(arr,i),j))

= SIZEL(LST(ASSIGN(arr,NEXT(arr,i,j-2)+4,ACCESS(arr,NEXT (arr, i,j-1)+4), i)

Observe que NEXT(arr,i,j-2)+4 é a posição, no Array arr, do elo do (j-1)-ésimo elemento de List. Nesta posição será atribuído o valor fornecido por ACCESS(arr,NEXT(arr,i,j-1)+4),isto é, o valor do elo do j-ésimo elemento (o elemento retirado da list). Portanto, o elemento que precedia o elemento retirado, precederá o sucessor do elemento retirado, ou seja, ao percorrer-se a list resultante, através da função NEXT, obter-se-á uma iteração a menos do que na Lista original.

Então, chamando de arr_1 ao Array resultante da operação ASSIGN(arr,NEXT(arr,i,j-2)+4,ACCESS(arr,NEXT(arr,i,j-1)+ 4), i), tem-se

SIZEL(DELIST(list,j))=SIZEL(LST(arr_1,i))

= SIZEL(LST(arr,i)) - 1

= SIZEL(list) - 1

Portanto, SIZEL(DELIST(list,j))=

IF j > SIZEL(list) V j = 0

THEN SIZEL(list)

ELSE SIZEL(list) - 1

SIZEL(JOIN(list,$list_1$,k))=

= SIZEL(JOIN(LST(arr,i),LST(arr_1,j),k)) (por definição)

Suponha:

1) k $\geq$ SIZEL(LST(arr,i))

Então, SIZEL(JOIN(LST(arr,i),LST(arr_1,j),k))=

= LST(LINK(ASSIGN(ASSIGN(arr,i+SIZEL(LST(arr_1,j))*5, i + SIZEL(LST(arr_1,j))*5+1),NEXT(arr,i,SIZEL(LST(arr,i))-1)+4,i+1),i,arr_1,j,(SIZEL(LST(arr_1,j))-1)*5), i + SIZEL(LST(arr_1,j)))

Observe que sobre o Array arr são feitos dois ASSIGN'S: o primeiro é na posição que ocupará o último elemento da lista resultante, atribuindo ao seu elo o endereço da próxima posição livre no Array resultante (i+SIZEL(LST(arr_1,j))*5+1), o segundo ASSIGN é feito sobre o elo do último elemento da List LST(arr,i), atribuindo a ele o valor i+1 que é o endereço do primeiro elemento da List LST(arr_1,j) a ser inserido na List LST(arr,i).

Chame de arr_2 ao Array resultante destes dois ASSIGN'S, isto é, arr_2 = ASSIGN(ASSIGN(arr,i+SIZEL(LST(arr_1,j))*5,i+SIZEL(LST(arr_1,j))*5+1),NEXT(arr,i,SIZEL(LST(arr,i))-1)+ 4, i+1).

A seguir, sobre arr_2 é aplicado o programa LINK com parâmetros (arr_2,i,arr_1,j,(SIZEL(LST(arr_1,j))-1)*5). Isto fará com que se tenha SIZEL(LST(arr_1,j)) aplicações do programa INSERTLM com parâmetros (arr_2,i,arr_1,j,m) onde m varia de 0 até SIZEL(LST(arr_1,j))-1)*5.

Cada INSERTLM irá atribuir valores a cinco elementos consecutivos de arr_2, onde os quatro primeiros são obtidos do acesso ao Array arr_1 e o quinto é o elo (i+m+6).

Portanto, a List resultante irá possuir todos os elementos de list e todos os elementos de $list_1$.

Logo, SIZEL(JOIN(LST(arr,i),LST(arr_1,j),k))=

= SIZEL(LST(arr,i)) + SIZEL(LST(arr_1,j))

2) k < SIZEL(LST(arr,i))

Se SIZEL(LST(arr_1,j)) = 0 então

SIZEL(JOIN(LST(arr,i), LST(arr_1,j),k))= SIZEL(LST(arr,i)) (programa L10)

= SIZEL(list)

= SIZEL(list) + 0

= SIZEL(list) + SIZEL($list_1$)

Se SIZEL(LST(arr_1,j)) ≠ 0 então

SIZEL(JOIN(LST(arr,i),LST(arr_1,j),k)) =

= SIZEL(LST(ASSIGN(ASSIGN(LINK(arr,i,arr_1,j,(SIZEL(LST(arr_1,
j))-1)*5),i+SIZEL(LST(arr_1,j))*5,NEXT(arr,i,k)),
NEXT(arr,i,k-1)+4,i+1)) (prog. L10)

Observe que a diferença entre este caso e aquele em que k ≥ SIZEL(LST(arr,i)) é que os ASSIGN'S são feitos após a aplicação do programa LINK. Isto é, o programa LINK copia os elementos de $list_1$ para list a partir da posição i+1 no Array arr e

a seguir é atribuído o endereço do elemento que seguia o k-ésimo elemento de list ao elo do último elemento de $list_1$ ao elo do k-ésimo elemento de list é atribuído o valor i+1, indicando que o elemento que o segue é o primeiro elemento de $list_1$.

Desta forma a List resultante terá os k primeiros elementos de list, seguidos dos elementos de $list_1$ e estes dos elementos de k+1 até SIZEL(list).

então SIZEL(JOIN(list,$list_1$,k))=SIZEL(list) + SIZEL($list_1$)

ELEMENT(CREATE,j)=ELEMENT(LST(NEWARRAY,0),j) (por def.)

= UNDEFINED pois SIZEL(LST(NEWARRAY,0),j) = 0

ELEMENT(INLIST(list,i,rtg_1),j)=ELEMENT(INLIST(LST(arr,k),i, rtg_1),j)

Suponha

1) i+1 = j e j < SIZEL(INLIST(LST(arr,k),i,rtg_1))

Então i+1 < SIZEL(INLIST(LST(arr,k),i,rtg_1))

→ i < SIZEL(LST(arr,k))

Chame arr_1 = ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(arr,k + 5,ACCESS(arr,NEXT(arr,k,i-1)+4)),NEXT(arr,k,i-1)+4, k+1),
k+1, ABSCISSAINF(rtg_1)),
k+2, ABSCISSASUP(rtg_1)),
k+3, ORDENADAINF(rtg_1)),
k+4, ORDENADASUP(rtg_1))

Então,

ELEMENT(INLIST(LST(arr,k),i,rtg_1),j)=ELEMENT(LST(arr_1,k+5),j)

Observe que:

NEXT(arr_1,k+5,j-1)=NEXT(arr_1,k,j-1) (por A1)

=NEXT(arr_1,k,i) (pois j=i+1)

=ACCESS(arr_1,NEXT(arr_1,k,i-1)+4) (por A2)

= k+1 (*)

Então por (*) e definição de arr_1

ACCESS(arr_1,NEXT(arr_1,k,j-1))=ACCESS(arr_1,k+1)=

=ABSCISSAINF(rtg_1)

ACCESS(arr_1,NEXT(arr_1,k,j-1)+1)=ACCESS(arr_1,k+2)=

=ABSCISSASUP(rtg_1)

ACCESS(arr_1,NEXT(arr_1,k,j-1)+2)=ACCESS(arr_1,k+3)=

=ORDENADAINF(rtg_1)

ACCESS(arr_1,NEXT(arr_1,k,j-1)+3)=ACCESS(arr_1,k+4)=

=ORDENADASUP(rtg_1) (**)

Logo:

ELEMENT(INLIST(LST(arr,k),i,rtg_1),j)=

= RTG(ABSCISSAINF(rtg_1),ABSCISSASUP(rtg_1),

ORDENADAINF(rtg_1),ORDENADASUP(rtg_1)) (prog. L5 (**))

= rtg_1 (***) (programas R7, R8, R9 e R10)

2) j ≠ i+1 e j < SIZEL(INLIST(LST(arr,k),i,rtg_1))

Seja arr_1 como definido acima e suponha i<SIZEL(LST(arr,k))

Então

ELEMENT(INLIST(LST(arr,k),i,rtg_1),j)=ELEMENT(LST(arr_1,k+5),j)

Seja NEXT(arr_1,k+5,j-1)=NEXT(arr,k,j-1) (pois j-1 < k e por A3)

Então ELEMENT(LST(arr_1,k+5,j)=ELEMENT(LST(arr,k),j)

=ELEMENT(list,j)

Por outro lado, se i = SIZEL(LST(arr,k)), tem-se

arr_1=ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(arr,k+5,k+6), k,

k+1),

k+1,ABSCISSAINF(rtg_1),

k+2,ABSCISSASUP(rtg_1),

k+3,ORDENADAINF(rtg_1),

k+4,ORDENADASUP(rtg_1))

Então,

ELEMENT(INLIST(LST(arr,k),i,rtg_1),j)=ELEMENT(LST(arr_1,k+5),j)

Observe que o elemento rtg_1 foi inserido no final da lista e portanto, para todo inteiro não negativo n, n < SIZEL (list)-1 tem-se

NEXT(arr,k,n)=NEXT(arr_1,k+5,n)

Logo ELEMENT(LST(arr_1,k+5),j)=ELEMENT(LST(arr,k),j)=

ELEMENT(list,j)

3) Suponha j > SIZEL(INLIST(LST(arr,k),i,rtg_1))

então ELEMENT(INLIST(LST(arr,k),i,rtg_1),j)=UNDEFINED (pelo programa L5)

Portanto,

ELEMENT(INLIST(list,i,rtg_1),j)=

IF j > SIZEL(list) + 1

THEN UNDEFINED

ELSE IF i+1 = j

THEN rtg_1

ELSE ELEMENT(list,j) (por 1,2 e 3)

ELEMENT(DELIST(list,i),j)

Sejam arr um Array e k um inteiro tal que

list=LST(arr,k)

ELEMENT(DELIST(list,i),j)=ELEMENT(DELIST(LST(arr,k),i),j)

1) Suponha i > SIZEL(LST(arr,k))

então DELIST(LST(arr,k),i)=LST(arr,k)

Portanto ELEMENT(DELIST(LST(arr,k),i),j)=

ELEMENT(LST(arr,k),j)

2) Suponha j > SIZEL(DELIST(LST(arr,k),i)

então ELEMENT(DELIST(LST(arr,k),i),j)=UNDEFINED (prog. L5)

3) Suponha j ≤ SIZEL(DELIST(LST(arr,k),i))) e i = j

então i ≤ SIZEL(DELIST(LST(arr,k),i)))

Chame arr_1 = ASSIGN(arr,NEXT(arr,k,i-2)+4,

ACCESS(arr,ACCESS(arr,NEXT(arr,i,j-2)+4)+4)

então

DELIST(LST(arr,k),i)=LST(arr_1,k)

Observe que i = j e que

NEXT(arr_1,k,j-1)=ACCESS(arr_1,NEXT(arr_1,k,j-2)+4) (A2)

=ACCESS(arr,ACCESS(arr,NEXT(arr,k,j-2)+4)+4)

(A3.e def. arr_1)

=ACCESS(arr,NEXT(arr,k,j-1)+4)

=NEXT(arr,k,j) (A2)

Logo, ELEMENT(LST(arr_1,k),j)=ELEMENT(LST(arr,k), j+

1) → ELEMENT(DELIST(LST(arr,k),i),j)=

ELEMENT(LST(arr,k),j+1)

4) Suponha $j < SIZEL(DELIST(LST(arr,k),i))$, $i \neq j$

e $i \leq SIZEL(LST(arr,k))$

Considere arr_1 como definido em (3)

$NEXT(arr_1,k,j-1)=ACCESS(arr_1,NEXT(arr_1,k,j-2)+4)$ (A2)

$=ACCESS(arr,NEXT(arr,k,j-2)+4)$ (def. de arr_1)

$=NEXT(arr,k,j-1)$

$ELEMENT(LST(arr_1,k),j)=ELEMENT(LST(arr,k),j)$

$ELEMENT(DELIST(LST(arr,k),i),j)=ELEMENT(LST(arr,k),j)$

Logo:

$ELEMENT(DELIST(list,i),j)=$

$=$ IF $j > SIZEL(DELIST(list,i))$

THEN UNDEFINED (caso 2)

ELSE IF $i = j$

THEN $ELEMENT(list,i+1)$ (caso 3)

ELSE $ELEMENT(list,j)$ (caso 1 e 4)

$ELEMENT(JOIN(list_1,list_2,i),j)$

Sejam arr e arr_1 Arrays e k, l inteiros tais que

$list_1 = LST(arr,k)$ e $list_2 = LST(arr_1,l)$.

1) Suponha $j > SIZEL(list_1) + SIZEL(list_2)$

$SIZEL(list_1)+SIZEL(list_2)=SIZEL(JOIN(list_1,list_2,i)$ (def. de SIZEL)

$ELEMENT(JOIN(LST(arr,k),LST(arr_1,l),i),j)=UNDEFINED$ (prog. L5)

2) Suponha $j \leq SIZEL(list_1) + SIZEL(list_2)$ e $j \leq i$

2.1) Suponha $i < SIZEL(LST(arr,k))$

Se $SIZEL(LST(arr_1,1)) = 0$ então

$JOIN(LST(arr,k),LST(arr_1,1),i)=LST(arr,k)$ (prog. L10)

então

$ELEMENT(JOIN(LST(arr,k),LST(arr_1,1),i),j)=ELEMENT(LST(arr,k),j)$

(pois $j < i$)

Se $SIZEL(LST(arr_1,1)) > 0$ então

$JOIN(LST(arr,k),LST(arr_1,1),i)=LST(ASSIGN(ASSIGN(LINK(arr,k,$
$arr_1,1,(SIZEL(LST(arr_1,1))-1$
$*5,\ k+SIZEL(LST(arr_1,1)*5,NEXT$
$(arr,k,i)),NEXT(arr,k,i-1)+4,k$
$+1)$

Observe que os elementos de $LST(arr_1,1)$ serão inseridos após a posição i e como $j \leq i$ tem-se que só serão atribuídos valores a elementos de arr com índice maior que k exceto na posição $NEXT(arr,k,i-1)+4$ que será atribuído o valor $NEXT(JOIN(LST(arr,k),LST(arr_1,1),i),k+SIZEL(LST(arr_1,1))*5,i)$. Então $NEXT(JOIN(LST(arr,k),LST(arr_1,1),i),k+SIZEL(LST(arr_1,1))*5,j-1) = NEXT(arr,k,j-1)$. Então $ELEMENT(JOIN(LST(arr,k),LST(arr_1,1),i),j) = ELEMENT(LST(arr,k),j)$

2.2) Suponha $i \geq SIZEL(LST(arr,k))$

$JOIN(LST(arr,k),LST(arr_1,1),i)=JOIN(LST(arr,k),LST($
$arr_1,1),SIZEL(LST($
$(arr,k))$

Analogamente ao caso 2.1 tem-se

ELEMENT(JOIN(LST(arr,k),LST(arr_1,1),i),j)=ELEMENT(LST(arr,k,j)

3) Suponha $j \leq SIZEL(list_1) + SIZEL(list_2)$, $i < j \leq i+SIZEL(list_2)$

Observe que:

O segundo ASSIGN do programa ATRIB(arr,k,arr_1,1,i) atualiza o elo do i-ésimo elemento e o programa LINK, através do programa INSERTELM copia a $list_2$ em arr a partir da posição i+1, logo o j-ésimo elemento estava originariamente em $list_2$.

Os primeiros elementos de arr continuam inalterados. O (j+1)-ésimo elemento será atribuído ao Array arr na posição k+1 (segundo o ASSIGN do programa ATRIB, o inicialmente j-ésimo elemento será o j+SIZEL($list_2$) (primeiro ASSIGN do programa ATRIB))

Então

NEXT(JOIN(LST(arr,k),LST(arr_1,1),i),k+SIZEL(LST(arr_1,1)),j-1) =
= NEXT(LST(arr_1,1),1,j-i-1)

ELEMENT(JOIN(LST(arr,k),LST(arr_1,1),i),j)=ELEMENT(LST(arr_1,1), j - i)

4) Suponha $j \leq SIZEL(list_1)+SIZEL(list_2)$ e $j > i+SIZEL(list_2)$

Analogamente ao caso 3

NEXT(JOIN(LST(arr,k),LST(arr_1,1),i),k+SIZEL(LST(arr_1,1)),j-1)=
= NEXT(LST(arr,k),k,j-SIZEL(LST(arr_1,1)) - 1)

ELEMENT(JOIN(LST(arr,k),LST(arr_1,1),i),j)=
= ELEMENT(LST(arr,k),j-SIZEL(LST(arr_1,1)))

Logo:

ELEMENT(JOIN($list_1$,$list_2$,i),j)=

= IF j > SIZEL($list_1$)+SIZEL($list_2$)

THEN UNDEFINED (caso 1)

ELSE IF j ≤ i

THEN ELEMENT($list_1$,j)) (caso 2)

ELSE IF j ≤ i + SIZEL($list_2$)

THEN ELEMENT($list_2$,j-1) (caso 3)

ELSE ELEMENT($list_1$,j-SIZEL($list_2$)) (caso 4)

Prova da Correção do Tipo Retang

Sejam r_1, r_2, r_3, r_4, s_1, s_2, s_3, s_4 inteiros tais que
$ret_1 = RTG(r_1, r_2, r_3, r_4)$ e $ret_2 = RTG(s_1, s_2, s_3, s_4)$
e i_1, j_1 inteiros tais que $pt = PT(i_1,j_1)$

então:

ISIN(ret_1,pt)=ISIN(RTG(r_1,r_2,r_3,r_4),PT(i_1,j_1) (pelo progr. RT12)

= $r_1 \leq i_1 \leq r_2$ $\wedge$ $r_3 \leq j_1 \leq r_4$ (pelos programas de RT7 a a RT10 e RT13 a RT14)

= ABSCISSAINF(RTG(r_1,r_2,r_3,r_4)) ≤ ABSCISSA(PT(i_1,j_1)) ≤ ABSCISSASUP(RTG(r_1,r_2,r_3,r_4))

$\wedge$ ORDENADAINF(RTG(r_1,r_2,r_3,r_4)) ≤ ORDENADA(PT(i_1,j_1)) ≤ ORDENADASUP(RTG(r_1,r_2,r_3,r_4))

= ABSCISSAINF(ret_1) ≤ ABSCISSA(pt) ≤ ABSCISSASUP(ret_1) $\wedge$
ORDENADAINF(ret_1) ≤ ORDENADA(pt) ≤ ORDENADASUP(ret_1)

$ISRETANG(ret_1) = ISRETANG(RTG(r_1,r_2,r_3,r_4))$ = (pelo prog. RT11)

$= r_1 \leq r_2 \quad \Lambda \quad r_3 \leq r_4$ (pelos programas de RT7 a RT10)

$= ABSCISSAINF(RTG(r_1,r_2,r_3,r_4)) \leq ABSCISSASUP(RTG(r_1,r_2,r_3,r_4)) \Lambda$

$ORDENADAINF(RTG(r_1,r_2,r_3,r_4)) \leq ORDENADASUP(RTG(r_1,r_2,r_3,r_4))$

$= ABSCISSAINF(ret_1) \leq ABSCISSASUP(ret_1) \Lambda ORDENADAINF(ret_1) \leq$

$ORDENADASUP(ret_1)$

$ret_1 - ret_2 = RTG(r_1,r_2,r_3,r_4) - RTG(s_1,s_2,s_3,s_4)$ = (pelo programa RT6)

$= INSERT(INSERT(INSERT(NEWLIST(RTG(r_1,s_1-1,r_3,r_4)),$

$RTG(s_1,s_2,r_3,s_3-1)), RTG(s_1,s_2,s_4+1,r_4)),$

$RTG(s_2+1,r_2,r_3,r_4))$ (1)

Note que $ABSCISSAINF(RTG(r_1,s_1-1,r_3,r_4)) = ABSCISSAINF(ret_1)$

$ABSCISSASUP(RTG(r_1,s_1-1,r_3,r_4)) = ABSCISSAINF(ret_2)-1$

$ORDENADAINF(RTG(r_1,s_1-1,r_3,r_4)) = ORDENADAINF(ret_1)$

$ORDENADASUP(RTG(r_1,s_1-1,r_3,r_3)) = ORDENADASUP(ret_1)$

$ABSCISSAINF(RTG(s_1,s_2,r_3,s_3-1)) = ABSCISSAINF(ret_2)$

$ABSCISSASUP(RTG(s_1,s_2,r_3,s_3-1)) = ABSCISSASUP(ret_2)$

$ORDENADAINF(RTG(s_1,s_2,r_3,s_3-1)) = ORDENADAINF(ret_1)$

$ORDENADASUP(RTG(s_1,s_2,r_3,s_3-1)) = ORDENADAINF(ret_2)-1$

$ABSCISSAINF(RTG(s_1,s_2,s_4+1,r_4)) = ABSCISSAINF(ret_2)$

$ABSCISSASUP(RTG(s_1,s_2,s_4+1,r_4)) = ABSCISSASUP(ret_2)$

$ORDENADAINF(RTG(s_1,s_2,s_4+1,r_4)) = ORDENADASUP(ret_2)+1$

$ORDENADASUP(RTG(s_1,s_2,s_4+1,r_4)) = ORDENADASUP(ret_1)$

ABSCISSAINF(RTG(s_2+1,r_2,r_3,r_4))=ABSCISSASUP(ret_2)+1

ABSCISSASUP(RTG(s_2+1,r_2,r_3,r_4))=ABSCISSASUP(ret_1)

ORDENADAINF(RTG(s_2+1,r_2,r_3,r_4))=ORDENADAINF(ret_1)

ORDENADASUP(RTG(s_2+1,r_2,r_3,r_4))=ORDENADASUP(ret_1)

Chame rtg_1 de RTG(r_1,s_1-1,r_3,r_4)

rtg_2 de RTG(s_1,s_2,r_3,s_3-1)

rtg_3 de RTG(s_1,s_2,s_4+1,r_4)

rtg_4 de RTG(s_2+1,r_2,r_3,r_4)

então a expressão (1) é igual a INSERT(INSERT(INSERT(NEWLIST(rtg_1), rtg_2), rtg_3), rtg_4) e como a expressão:

ABSCISSAINF(rtg_1)=ABSCISSAINF(ret_1) ∧ ABSCISSASUP(rtg_1)=
ABSCISSAINF(ret_2) - 1 ∧
ORDENADAINF(rtg_1)=ORDENADAINF(ret_1) ∧ ORDENADASUP(rtg_1)=
ORDENADASUP(ret_1) ∧
ABSCISSAINF(rtg_2)=ABSCISSAINF(ret_2) ∧ ABSCISSASUP(rtg_2)=
ABSCISSASUP(ret_2) ∧
ORDENADAINF(rtg_2)=ORDENADAINF(ret_1) ∧ ORDENADASUP(rtg_2)=
ORDENADAINF(ret_2)-1 ∧
ABSCISSAINF(rtg_3)=ABSCISSAINF(ret_2) ∧ ABSCISSASUP(rtg_3)=
ABSCISSASUP(ret_2) ∧
ORDENADAINF(rtg_3)=ORDENADASUP(ret_2)+1∧ORDENADASUP(rtg_3)=
ORDENADASUP(ret_1) ∧
ABSCISSAINF(rtg_4)=ABSCISSASUP(ret_2)+1∧ABSCISSASUP(rtg_4)=
ABSCISSASUP(ret_1) ∧
ORDENADAINF(rtg_4)=ORDENADAINF(ret_1) ∧ ORDENADASUP(rtg_4)=
ORDENADASUP(ret_1)

é TRUE temos finalmente que a expressão (1) é igual a

IF ABSCISSAINF(rtg_1)=ABSCISSAINF(ret_1) $\wedge$ ABSCISSASUP(rtg_1)=
ABSCISSAINF(ret_2)-1
$\wedge$ ORDENADAINF(rtg_1)=ORDENADAINF(ret_1) $\wedge$ ORDENADASUP(rtg_1)=
ORDENADASUP(ret_1)
$\wedge$ ABSCISSAINF(rtg_2)=ABSCISSAINF(ret_2) $\wedge$ ABSCISSASUP(rtg_2)=
ABSCISSASUP(ret_2)
$\wedge$ ORDENADAINF(rtg_2)=ORDENADAINF(ret_1) $\wedge$ ORDENADASUP(rtg_2)=
ORDENADAINF(ret_2)-1
$\wedge$ ABSCISSAINF(rtg_3)=ABSCISSAINF(ret_2) $\wedge$ ABSCISSASUP(rtg_3)=
ABSCISSASUP(ret_2)
$\wedge$ ORDENADAINF(rtg_3)=ORDENADASUP(ret_2) $\wedge$ ORDENADASUP(rtg_3)=
ORDENADASUP(ret_1)
$\wedge$ ABSCISSAINF(rtg_4)=ABSCISSASUP(ret_2)+1 $\wedge$ ABSCISSASUP(rtg_4)=
ABSCISSASUP(ret_1)
$\wedge$ ORDENADAINF(rtg_4)=ORDENADAINF(ret_1) $\wedge$ ORDENADASUP(rtg_4)=
ORDENADASUP(ret_1)

THEN INSERT(INSERT(INSERT(NEWLIST(rtg_1), rtg_2), rtg_3), rtg_4)

ret_1 = RTG(r_1,r_2,r_3,r_4) e ret_2 = RTG(s_1,s_2,s_3,s_4)

então

ABSCISSAINF(ret_1)=r_1 ABSCISSAINF(ret_2)=s_1 (pròg. RT7)
ABSCISSASUP(ret_1)=r_2 ABSCISSASUP(ret_2)=s_2 (prog. RT8)
ORDENADAINF(ret_1)=r_3 ORDENADAINF(ret_2)=s_3 (prog. RT9)
ORDENADASUP(ret_1)=r_4 ORDENADASUP(ret_2)=s_4 (prog. RT10)

Para provar que o programa RT5 implementa a intersecção deve-se mostrar que ele satisfaz os 4 axiomas que definem a intersecção:

1) ABSCISSAINF($ret_1 \cap ret_2$) 2) ABSCISSASUP($ret_1 \cap ret_2$)

3) ORDENADAINF($ret_1 \cap ret_2$) 4) ORDENADASUP($ret_1 \cap ret_2$)

Então

1) ABSCISSAINF($ret_1 \cap ret_2$) =

= ABSCISSAINF(RTG(r_1,r_2,r_3,r_4) $\cap$ RTG(s_1,s_2,s_3,s_4))

Então se ISRETANG(RTG(r_1,r_2,r_3,r_4)) $\wedge$ ISRETANG(RTG(s_1,s_2,s_3,s_4)) é falso, examinando-se a linha 43 do programa RT5, que corresponde ao ELSE do IF da linha 1 observa-se que o resultado é um Retang onde o primeiro elemento da quádrupla é o valor 1 o que prova que

ABSCISSAINF($ret_1 \cap ret_2$) = 1 se ISRETANG(ret_1) $\wedge$ ISRETANG(ret_2) é falso. (*1)

Supor que ISRETANG(RTG(r_1,r_2,r_3,r_4)) $\wedge$ ISRETANG(RTG(s_1,s_2,s_3,s_4)) é verdadeiro e além disto:

i) $r_1 \leq s_1 \leq r_2 \quad \wedge \quad (r_3 \leq s_3 \leq r_4 \quad \vee \quad s_3 \leq r_3 \leq s_4)$

No programa RT5, as linhas que satisfazem esta condição são 6, 7, 10, 11, 15, 16, 19 e 20 e pode-se observar que em todas elas o primeiro elemento da quádrupla é s_1 = ABSCISSAINF(ret_2) o que prova que ABSCISSAINF($ret_1 \cap ret_2$) = ABSCISSAINF(ret_2) se ABSCISSAINF(ret_1) $\leq$ ABSCISSAINF(ret_2) $\leq$ ABSCISSAINF(ret_1) é verdadeiro e se ORDENADAINF(ret_1) $\leq$ ORDENADAINF(ret_2) $\leq$ ORDENADASUP(ret_1) $\vee$ ORDENADAINF(ret_2) $\leq$ ORDENADAINF(ret_1) $\leq$ ORDENADASUP(ret_2) também é verdadeiro.

ii) $r_1 \leq s_1 \leq r_2 \quad \wedge \quad \neg (r_3 \leq s_3 \leq r_4 \quad \vee \quad s_3 \leq r_3 \leq s_4)$

Pode-se observar que esta condição é satisfeita pe-

las linhas 12 e 21 do programa RT5 o que mostra que ABSCISSAINF $(ret_1 \cap ret_2)=1$ se ABSCISSAINF$(ret_1) \leq$ ABSCISSAINF$(ret_2) \leq$ ABSCISSASUP(ret_1) é verdadeiro mas falha a condição (ORDENADAINF $(ret_1) \leq$ ORDENADAINF$(ret_2) \leq$ ORDENADASUP(ret_1) V ORDENADAINF $(ret_2) \leq$ ORDENADAINF$(ret_1) \leq$ ORDENADASUP(ret_2)).

iii) $s_1 \leq r_1 \leq s_2 \ \wedge \ (r_3 \leq s_3 \leq r_4 \ \vee \ s_3 \leq r_3 \leq s_4)$

No programa RT5, as linhas que satisfazem esta condição são as linhas 26, 27, 30, 31, 35, 36, 39 e 40 e pode-se observar que em todas elas o primeiro elemento da quádrupla é r_1= ABSCISSAINF(ret_1), o que prova que ABSCISSAINF$(ret_1 \quad ret_2)$= ABSCISSAINF(ret_1) se ABSCISSAINF$(ret_2) \leq$ ABSCISSAINF$(ret_1) \leq$ ABSCISSASUP(ret_2) é verdadeiro e se ORDENADAINF$(ret_1) \leq$ ORDENADAINF$(ret_2) \leq$ ORDENADASUP(ret_1) V ORDENADAINF$(ret_2) \leq$ ORDENADAINF$(ret_1) \leq$ ORDENADASUP(ret_2) também é verdadeiro.

iv) $s_1 \leq r_1 \leq s_2 \quad \wedge \neg (r_3 \leq s_3 \leq r_4 \quad \vee s_3 \leq r_3 \leq s_4)$

Esta condição é satisfeita pelas linhas 32 e 41 do programa RT5 o que mostra que ABSCISSAINF$(ret_1 \cap ret_2)=1$ se ABSCISSAINF$(ret_2) \leq$ ABSCISSAINF$(ret_1) \leq$ ABSCISSASUP(ret_2) é verdadeiro mas falha a condição (ORDENADAINF$(ret_1) \leq$ ORDENADAINF $(ret_2) \leq$ ORDENADASUP(ret_1) V ORDENADAINF$(ret_2) \leq$ ORDENADAINF $(ret_1) \leq$ ORDENADASUP(ret_2))

v) $\neg (r_1 \leq s_1 \leq r_2) \quad \wedge \quad \neg (s_1 \leq r_1 \leq s_2)$

Esta condição é satisfeita pela linha 42 do programa

RT5 e o primeiro elemento da quádrupla é o valor 1 o que mostra que ABSCISSAINF($ret_1 \cap ret_2$)=1 se ABSCISSAINF(ret_1) $\leq$ ABSCISSAINF(ret_2) $\leq$ ABSCISSASUP(ret_1) é falso e ABSCISSAINF (ret_2) $\leq$ ABSCISSAINF(ret_1) $\leq$ ABSCISSASUP(ret_2) também é falso.

De (*1), i, ii, iii, iv e v resulta que

ABSCISSAINF($ret_1 \cap ret_2$) =

= IF ISRETANG(ret_1) $\wedge$ ISRETANG(ret_2)

THEN IF ABSCISSAINF(ret_1) $\leq$ ABSCISSAINF(ret_2) $\leq$ ABSCISSASUP(ret_1)

THEN IF ORDENADAINF(ret_1) $\leq$ ORDENADAINF(ret_2) $\leq$ ORDENADASUP(ret_1) V ORDENADAINF(ret_2) $\leq$ ORDENADAINF(ret_1) $\leq$ ORDENADASUP(ret_2)

THEN ABSCISSAINF(ret_2) (i)

ELSE 1 (ii)

ELSE IF ABSCISSAINF(ret_2) $\leq$ ABSCISSAINF(ret_1) $\leq$ ABSCISSASUP(ret_2)

THEN IF ORDENADAINF(ret_1) < ORDENADAINF (ret_2) < ORDENADASUP(ret_1) V ORDENADAINF(ret_2) < ORDENADAINF (ret_1) < ORDENADASUP(ret_2)

THEN ABSCISSAINF(ret_1) (iii)

ELSE 1 (iv)

ELSE 1 (v)

ELSE 1 (*1)

Analogamente demonstra-se que o programa RT5 satisfaz os axiomas ABSCISSASUP($ret_1 \cap ret_2$), ORDENADAINF($ret_1 \cap ret_2$) e ORDENADASUP($ret_1 \cap ret_2$).

CORREÇÃO DA RETANGLIST

Sejam

- retlist uma Retanglist
- r_1, r_2, r_3, r_4, inteiros tais que $rtg_1 = RTG(r_1, r_2, r_3, r_4)$
- i_1, j_1, inteiros tais que $pt = PT(i_1, j_1)$
- arr um Array e i, j inteiros positivos.

então:

$ISINL(NEWLIST(rtg_1), pt) = ISINL(NEWLIST(RTG(r_1, r_2, r_3, r_4)), pt)$ (definição rtg_1)

Suponha $ISRETANG(rtg_1) = TRUE$ então $ISRETANG(RTG(r_1, r_2, r_3, r_4)) =$ TRUE o que implica

$ISINL(NEWLIST(RTG(r_1, r_2, r_3, r_4)), PT(i_1, j_1)) =$

$= ISINL(RTLIST(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(NEWARRAY, 1, +), 2, r_1), 3, r_2), 4, r_3), 5, r_4), 5), PT(i_1, j_1))$ (pelo programa RT1)

Como $i = 5 (\neq 0)$ e $ACCESS(arr, i-4) = + (-)$,

chamando $ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(NEWARRAY, 1, +), 2, r_1), 3, r_2), 4, r_3), 5, r_4)$ de arr e

$ACCESS(arr, 5-3) = r_1$, $ABSCISSA(PT(i_1, j_1)) = i_1$ (prog. RT13)

$ACCESS(arr, 5-2) = r_2$, $ORDENADA(PT(i_1, j_1)) = j_1$ (prog. RT14)

$ACCESS(arr, 5-1) = r_3$ e

$ACCESS(arr, 5) = r_4$

então pelo programa 4

Se 1) $r_1 \leq i_1 \leq r_2 \quad \wedge \quad r_3 \leq j_1 \leq r_4 = TRUE =$

$= ISIN(RTG(r_1, r_2, r_3, r_4), PT(i_1, j_1))$ (programa RT12)

$= ISIN(rtg_1, pt)$ (pela definição)

2) $r_1 \leq i_1 \leq r_2 \quad \Lambda \quad r_3 \leq j_1 \leq r_4$ = FALSE então tem-se

ISINL(RTLIST(arr,i-5),pt) (programa RT4)

=ISINL(RTLIST(arr,0),PT(i_1,j_1)) (por definição)

=FALSE (i=0 programa RT4)

=$r_1 \leq i_1 \leq r_2 \quad \Lambda \quad r_3 \leq j_1 \leq r_4$ = ISIN(RTG(r_1,r_2,r_3,r_4), PT(i_1,j_1))

Então conclui-se que se rtg_1 é um Retang tem-se

ISINL(NEWLIST(rtg_1),pt)=ISIN(rtg_1,pt)

Suponha, agora, que ISRETANG(rtg_1)=FALSE então

ISINL(NEWLIST(rtg_1),pt)=ISINL(RTLIST(NEWARRAY,0),pt) (pelo programa RT1)

= FALSE (pelo programa 4, pois i = 0)

logo

ISINL(NEWLIST(rtg_1),pt)= IF ISRETANG(rtg_1)
THEN ISIN(rtg_1,pt)
ELSE FALSE

ISINL(INSERT(retlist,rtg_1),pt)=

= ISINL(INSERT(RTLIST(arr,i),RTG(r_1,r_2,r_3,r_4)),PT(i_1,j_1))

Suponha que:

(1) ISRETANG(RTG(r_1,r_2,r_3,r_4)) = TRUE então tem-se

ISINL(INSERT(RTLIST(arr,i),RTG(r_1,r_2,r_3,r_4)), PT(i_1,j_1))=

= ISINL(RTLIST(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(arr,i+1,+), i+2,r_1),i+3,r_2),i+4,r_3),i+5,r_4),i+5), PT(i_1,j_1))

chamando ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(arr,i+1,+),i+2,r_1),i+3,r_2),i+4,r_3),i+5,r_4) de arr_1

tem-se

$i \neq 0$

$ACCESS(arr_1,(i+5)-4) \neq -$

$ACCESS(arr_1,(i+5)-3) = r_1$

$ACCESS(arr_1,(i+5)-2) = r_2$

$ACCESS(arr_1,(i+5)-1) = r_3$

$ACCESS(arr_1,(i+5)) = r_4$

$ABSCISSA(PT(i_1,j_1))=i_1$ (prog. RT13)

$ORDENADA(PT(i_1,j_1))=j_1$ (prog. RT14)

então se

$$r_1 \leq i_1 \leq r_2 \quad \Lambda \quad r_3 \leq j_1 \leq r_4 = TRUE$$

então

$ISINL(RTLIST(arr_1,i+5),pt)=TRUE=ISIN(rtg_1,pt)$ (*1)

Se $r_1 \leq i_1 \leq r_2 \quad \Lambda \quad r_3 \leq j_1 \leq r_4$=FALSE

então, pelo programa RT4

$ISINL(RTLIST(arr_1,i+5),pt)=ISINL(RTLIST(arr_1 i+5-5,pt)$

$= ISINL(RTLIST(arr,i),pt)=ISINL(retlist,pt)$ (*2)

Por (*1) e (*2) segue

$ISINL(INSERT(retlist,rtg_1),pt)=ISINL(retlist,pt) \ V \ ISIN(rtg_1,pt)$

2) $ISRETANG(RTG(r_1,r_2,r_3,r_4)) = FALSE$

então tem-se

$ISINL(INSERT(RTLIST(arr,i),RTG(r_1,r_2,r_3,r_4)),PT(i_1,j_1))=$

$= ISINL(RTLIST(arr,i),PT(i_1,j_1))$

logo

ISINL(INSERT(retlist,rtg_1),pt)=

= IF ISRETANG(rtg_1)

THEN ISINL(retlist,pt) V ISIN(rtg_1,pt)

ELSE ISINL(retlist,pt)

Analogamente prova-se que

ISINL(DELETE(retlist,rtg_1),pt)=

= IF ISRETANG(rtg_1)

THEN ISINL(retlist,pt) $\wedge$ $\neg$ ISIN(rtg_1,pt)

ELSE ISINL(retlist,pt)

SIZE(NEWLIST(rtg_1))=SIZE(NEWLIST(RTG(r_1,r_2,r_3,r_4)))

Suponha que (1) ISRETANG(RTG(r_1,r_2,r_3,r_4)) = TRUE

então

SIZE(NEWLIST(RTG(r_1,r_2,r_3,r_4))) =

=SIZE(RTLIST(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(NEWARRAY, 1, +),2,r_1),3,r_2),4,r_3),5,r_4),5))

chamando ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(NEWARRAY,1,+),2,r_1),3,r_2),4,r_3),5,r_4) de arr e

substituindo-se tem-se

SIZE(NEWLIST(RTG(r_1,r_2,r_3,r_4))) = SIZE(RTLIST(arr,5))

pelo programa 15

SIZE(RTLIST(arr,5))=DIV(5,5) = 1

2) ISRETANG(RTG(r_1,r_2,r_3,r_4)) = FALSE

então

SIZE(NEWLIST(RTG(r_1,r_2,r_3,r_4))) = SIZE(RTLIST(NEWARRAY,0)) (pelo programa RT1)

= DIV(0,5) = 0 (pelo programa 15)

Logo

SIZE(NEWLIST(rtg_1)=IF ISRETANG(rtg_1)
THEN 1
ELSE 0

SIZE(INSERT(retlist,rtg_1))=
=SIZE(INSERT(RTLIST(arr,i),RTG(r_1,r_2,r_3,r_4)))

Suponha que:

(1) ISRETANG(RTG(r_1,r_2,r_3,r_4)) = TRUE

então

SIZE(INSERT(RTLIST(arr,i),RTG(r_1,r_2,r_3,r_4)))=
=SIZE(RTLIST(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(arr,i+1,+),
i+2,r_1),i+3,r_2),i+4,r_3),i+5,r_4),i+5)) (prog. RT2)
=DIV(i+5,5) (prog. RT15)
=DIV(i,5) + 1
=SIZE(RTLIST(arr,i)) + 1
=SIZE(retlist) + 1

(2) ISRETANG(RTG(r_1,r_2,r_3,r_4)) = FALSE

então

SIZE(INSERT(RTLIST(arr,i),RTG(r_1,r_2,r_3,r_4)))=
=SIZE(RTLIST(arr,i)) (prog. RT2)
=SIZE(retlist)

Logo

SIZE(INSERT(retlist,rtg_1)=

IF ISRETANG(rtg_1)

THEN SIZE(retlist) + 1

ELSE SIZE(retlist)

Analogamente prova-se que:

SIZE(DELETE(retlist,rtg_1))=

IF ISRETANG(rtg_1)

THEN SIZE(retlist) + 1

ELSE SIZE(retlist)

RETURN(NEWLIST(rtg_1),i)=RETURN(NEWLIST(RTG(r_1,r_2,r_3,r_4),i))

Suponha que:

(1) ISRETANG(RTG(r_1,r_2,r_3,r_4)) = TRUE

então

RETURN(NEWLIST(RTG(r_1,r_2,r_3,r_4),i))=

=RETURN(RTLIST(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(NEWARRAY,

1,+),2,r_1),3,r_2),4,r_3),5,r_4),5),i)) (prog. RT1)

Chamando ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(NEWARRAY,1,+),2,r_1),

3,r_2),4,r_3),5,r_4) de arr temos

RETURN(NEWLIST(RTG(r_1,r_2,r_3,r_4),i))=

= RETURN(RTLIST(arr,5),i)

Se i > DIV(5,5) então RETURN(RTLIST(arr,5),i)=

= UNDEFINED (*3) (prog. RT16)

Se i = DIV(5,5) então RETURN(RTLIST(arr,5),i)=

= RTG(ACCESS(arr,i-3),ACCESS(arr,i-2), ACCESS

(arr,i-1),ACCESS(arr,i)) (prog. RT16)

= RTG(r_1,r_2,r_3,r_4) = rtg_1 (*4)

Como DIV(5,5) = 1 não pode ocorrer i < DIV(5,5) e portanto, o ELSE da última linha do programa 16 nunca será atingido.

(2) ISRETANG(RTG(r_1,r_2,r_3,r_4)) = FALSE

então

RETURN(NEWLIST(RTG(r_1,r_2,r_3,r_4)),i)=RETURN(RTLIST(NEWARRAY,0),i)

como sempre i > DIV(0,5) então

RETURN(RTLIST(NEWARRAY,0),i) = UNDEFINED (*5)

Logo, por (*3), (*4), (*5) tem-se

RETURN(NEWLIST(rtg_1),i) = IF ISRETANG(rtg_1)

THEN IF i = 1

THEN rtg_1

ELSE UNDEFINED

ELSE UNDEFINED

RETURN(INSERT(retlist,rtg_1),i)=

=RETURN(INSERT(RTLIST(arr,j),RTG(r_1,r_2,r_3,r_4)),i)

Suponha que

(1) ISRETANG(RTG(r_1,r_2,r_3,r_4)) = TRUE

então

RETURN(INSERT(RTLIST(arr,j),RTG(r_1,r_2,r_3,r_4)),i)=

=RETURN(RTLIST(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(arr,j+1,

+),j+2,r_1),j+3,r_2),j+4,r_3),j+5,r_4),j+5),i)

chamando ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(arr,j+1,+),j+2, r_1),

j+3,r_2),j+4,r_3),j+5,r_4) de arr_1 e substituindo,

vem:

RETURN(INSERT(RTLIST(arr,j),RTG(r_1,r_2,r_3,r_4)),i)=

= RETURN(RTLIST(arr_1,j+5),i) (*6)

Se i > DIV(j+5,5)=SIZE(retlist) + 1 (pelo programa 15)

então RETURN(RTLIST(arr_1,j+5),i)=UNDEFINED (prog. RT16)

Se i = DIV(j+5,5) então RETURN(RTLIST(arr_1,j+5),i)=

=RTG(ACCESS(arr_1,j+5-3),ACCESS(arr_1,j+5-2),ACCESS(arr_1,j+ 5-

1),ACCESS(arr_1,j+5))=RTG(r_1,r_2,r_3,r_4) (programa RT16)

(*7)

Se i < DIV(j+5,5) então RETURN(RTLIST(arr_1,j+5),i)=

= RETURN(RTLIST(arr,j),i) (*8)

(2) ISRETANG(RTG(r_1,r_2,r_3,r_4)) = FALSE

então

RETURN(INSERT(RTLIST(arr,j),RTG(r_1,r_2,r_3,r_4)),i)=

=RETURN(RTLIST(arr,j),i) (*9)

Logo,

RETURN(INSERT(retlist,rtg_1),i)=

IF ISRETANG(rtg_1)

THEN IF i > SIZE(retlist) + 1

THEN UNDEFINED (de (*6))

ELSE IF i = SIZE(retlist) + 1

THEN rtg_1 (de (*7))

ELSE RETURN(retlist,i) (de (*8))

ELSE RETURN(retlist,i) (de (*9))

Analogamente, mostra-se que

RETURN(DELETE(retlist,rtg_1),i)=

= IF ISRETANG(rtg_1)

THEN IF i > SIZE(retlist) + 1

THEN UNDEFINED

ELSE IF i = SIZE(retlist) + 1

THEN rtg_1

ELSE RETURN(retlist,i)

ELSE RETURN(retlist,i)

RETSIG(NEWLIST(rtg_1),j)=RETSIG(NEWLIST(RTG(r_1,r_2,r_3,r_4)),j)

Supor que ISRETANG(rtg_1) = TRUE então

RETSIG(NEWLIST(RTG(r_1,r_2,r_3,r_4)),j)=RETSIG(RTLIST(ASSIGN(ASSIGN(

ASSIGN(ASSIGN(ASSIGN(NEWARRAY,1,+),2,r_1),3,r_2),4,r_3), 5,

r_4),5),j)

Se j ≠ 1 então j = 0 V j*5 > 5 e tem-se

RETSIG(NEWLIST(rtg_1),j)=UNDEFINED (*1)

Se j = 1 então

RETSIG(NEWLIST(rtg_1),j)=

=ACCESS(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(NEWARRAY,1,+), 2,

r_1),3,r_2),4,r_3),5,r_4),5),1)

= + (*2)

Por outro lado, se ISRETANG(rtg_1) = FALSE tem-se

RETSIG(NEWLIST(rtg_1),j)=RETSIG(RTLIST(NEWARRAY,0),j)=

= ACCESS(NEWARRAY,j)

= UNDEFINED (*3)

De (*1), (*2), (*3) resulta que

RETSIG(NEWLIST(rtg_1),j)=

=IF ISRETANG (rtg_1)

THEN IF j = 1

THEN + (de *2)

ELSE UNDEFINED (de *1)

ELSE UNDEFINED (de *3)

RETSIG(INSERT(retlist,rtg_1),j)=

=RETSIG(INSERT(RTLIST(arr,i),RTG(r_1,r_2,r_3,r_4)),j)

Suponha ISRETANG(rtg_1)= TRUE então

RETSIG(INSERT(RTLIST(arr,i),RTG(r_1,r_2,r_3,r_4)),j) =

=RETSIG(RTLIST(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(arr,i+1,+),

i+2,r_1),i+3,r_2),i+4,r_3),i+5,r_4)i+5),j)

Note que SIZE(retlist)=DIV(i,5) então

Se j = DIV(i,5)+1 então j = SIZE(retlist) + 1

e tem-se j*5=(DIV(i,5)+1)*5 = DIV(i,5)*5+5 = i+5

e tem-se

RETSIG(INSERT(retlist,rtg_1),j)=

=ACCESS(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(arr,i+1,+), i+ 2,

r_1),i+3,r_2),i+4,r_3),i+5,r_4),i+5),i+5-4)

= +

Por outro lado, se j > SIZE(retlist) + 1 tem-se

j > SIZE(retlist) + 1 e

RETSIG(INSERT(retlist,rtg_1),j)=

= RETSIG(RTLIST(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(arr, i+1, +), i+2,r_1), i+3,r_2),i+4,r_3),i+5,r_4),i+5),j)

= UNDEFINED pois se j > DIV(i,5) + 1 então

j * 5 > (DIV(i,5)+1)*5

j * 5 > (DIV(i,5)*5+5

j * 5 > i + 5

Supor j < SIZE(retlist) + 1 então j < DIV(i,5)+1 e ainda j * 5 < i + 5

Se j = 0 então fazendo arr_1=ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(arr,i+1,+),i+2,r_1),i+3,r_2),i+4,r_3),i+5,r_4)

tem-se

RETSIG(INSERT(retlist,rtg_1),0)=

= RETSIG(RTLIST(arr_1,i+5),0)

= ACCESS(arr_1,0)

= UNDEFINED = RETSIG(retlist,j)

Se j = 0 tem-se

RETSIG(INSERT(retlist,rtg_1),j)=

= ACCESS(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(arr,i+1,+), i+2,r_1),i+3,r_2),i+4,r_3),i+5,r_4),i+5),j*5-4)

= ACCESS(arr,j*5-4)= RETSIG(retlist,j)

Supor j > SIZE(retlist) então j > DIV(i,5) → j*5 > i → RETSIG(INSERT(retlist,rtg_1),j)=UNDEFINED

Supor j < SIZE(retlist) → j < DIV(i,5) → j*5 < i

j = 0 já foi estudado.

$j \neq 0 \rightarrow$ RETSIG(INSERT(retlist,rtg_1),j)=

= ACCESS(ASSIGN(ASSIGN(ASSIGN(ASSIGN(ASSIGN(arr,i+1,+), i+2,

r_1),i+3,r_2),i+4,r_3),i+5,r_4),i+5), j*5-4)=

= ACCESS(arr,j*5-4)

= RETSIG(retlist,j)

Analogamente demonstra-se que

RETSIG(DELETE(retlist,rtg_1),j)=

```
= IF ISRETANG(rtg1)
      THEN IF j = SIZE(retlist) + 1
                THEN -
                ELSE IF j >  SIZE(retlist) + 1
                          THEN UNDEFINED
                          ELSE RETSIG(retlist,j)
      ELSE IF j > SIZE(retlist)
                THEN UNDEFINED
                ELSE RETSIG(retlist,j)
```

APÊNDICE 3 - EXPLICAÇÕES SOBRE ALGUNS TERMOS UTILIZADOS

EXPLICAÇÕES SOBRE ALGUNS TERMOS UTILIZADOS

Álgebra - é um par [C,F], onde C é um conjunto não vazio de valores e F é um conjunto finito de operações com um número finito de argumentos, $f_i: C^n \rightarrow C$.

Álgebra heterogênea - é um par [V,F] onde V é um conjunto de conjuntos não vazios V_i e F um conjunto finito de operações com um número finito de argumentos $f_i: Vi_1 \times Vi_2 \times \ldots \times Vi_n \rightarrow V_h$ onde $\forall\, 1 \leq h \leq n \;\; [V_{i_h} \in V] \wedge V_h \in V$.

Fórmula Bem Formada - em uma especificação, existem regras que diferem na aplicação de operações, por exemplo, a primeira opeção a ser aplicada deve ser uma operação de criação. Uma seqüência de operações que satisfaçam estas regras é chamada uma fórmula bem formada.

Homomorfismo - é uma mapeamento de um sistema algébrico em outro sistema algébrico que preserva a estrutura, isto é, sejam

$(S, \oplus)$ e $(S', \otimes)$ dois sistemas algébricos

$h: S \rightarrow S'$ é um homomorfismo sss

$\forall\, a, b \in S, \; h(a \oplus b) = h(a) \otimes h(b)$

Isomorfismo - é um homomorfismo biunívoco, isto é, se

$f: S \rightarrow S'$ é um homomorfismo e f é biunívoco, então

f é um isomorfismo.

Objetos que são isomorfos são matematicamente indistinguíveis, portanto, propriedades conhecidas para um são carregadas imediatamente sobre o outro.

Modelo - uma interpretação para uma especificação de um tipo abstrato é um modelo se toda fórmula bem formada satisfaz a semântica da especificação.

Recursão mútua - o princípio de indução pode ser usado para definir um objeto, então se diz que o objeto é definido indutivamente ou recursivamente, por exemplo:

$a \times 0 = 0$

$a \times (b+1) = a \times b + b$

Mas se duas funções são definidas como:

$f_1 (x,0) = g_1(x)$

$f_2 (x,0) = g_2(x)$

$f_1 (x, y+1) = h_1 [x, y, f_1(x,y), f_2(x,y)]$

$f_2 (x, y+1) = h_2 [x, y, f_1(x,y), f_2(x,y)]$

se diz que f_1 e f_2 são definidas por recursão mútua.

Tipo primitivo - um tipo abstrato é dito um tipo primitivo se não é implementado em função de outro. Por exemplo, nesse trabalho o array foi considerado um tipo primitivo.

APENDICE 4 - INDICES DOS NOMES DE OPERAÇÕES, FUNÇÕES E REPRESENTAÇÕES USADOS NO DESENVOLVIMENTO DO EXEMPLO.

ÍNDICE DOS NOMES DE OPERAÇÕES, FUNÇÕES E REPRESENTAÇÕES USADOS NO DESENVOLVIMENTO DO EXEMPLO

Com o objetivo de facilitar a leitura do capítulo 5 e dos apêndices 1 e 2 é fornecido, neste apêndice, um índice dos termos utilizados na especificação e implementação dos tipos abstratos de dados Grid, Delgrid, Retanggr, Point, Retang, Retanglist, List e Array.

Na coluna "código" utiliza-se a seguinte convenção:

OP para identificar uma operação

FM para identificar uma função de mapeamento

FA para identificar uma função auxiliar

Cada tipo abstrato de dados será identificado por duas letras conforme a convenção dada a seguir:

GR → Grid

DG → Delgrid

RG → Retanggr

PT → Point

RT → Retang

RL → Retanglist

LT → List

AR → Array

assim, um código dado por OP/GR irá identificar uma operação (OP) sobre o tipo Grid (GR).

Os números das páginas fornecidas para cada termo dizem respeito ao local onde pode ser encontrada a sua definição (informal e formal) e a sua implementação.

7 BIBLIOGRAFIA

1. CARVALHO, R.L., et alii. A Model theoretic approach to the semantics of data types and structures. s.n.t.

2. EARLEY, J. Toward and understanding of data structures. Communications of the ACM, New York, 14(10):617-27, Oct. 1971.

3. FLECK, A.C. Recent developments in the theory of data structures. Computer Languages, Oxford, 3(1):37-52,1978.

4. GEHANI, N. A high level data structure-The Grid. Computer Languages, Oxford, 4(2);93-8, 1979.

5. GELLER, M. Test data as an aid in proving programa correcte ness. Communications of the ACM, New York, 21(5):368-75, May 1978.

6. GOGUEN, J.A.; THATCHER, J.W. & WAGNER, E.G. An initial algebra approach to the specification, correcteness and implementation of abstract data types. In: YEH, R.T. Current trends in programming methodology. Englewood Cliffs, Prentice Hall, 1978. v. 4, cap. 5, p. 81-149.

7. GRIES, D. & GEHANI, N. Some ideas on data types in high -level languages. Communications of the ACM, New York, 20(6):414-20, June 1977.

8. GUTTAG, J.V. Abstract data types and the development of data structures. Communications of the ACM, New York, 20 (6):396-404, June 1977.

9. _____. Notes on type abstraction (version 2). IEEE Transactions on Software Engineering, New York, SE-6 (1): 13-23, Jan. 1980.

10. _____ & HORNING, J.J. The algebraic specification of abstract data types. Acta Informatica, Berlin, 10(1):27-52, 1978.

11. _____; HOROWITZ, E. & MUSSER, D.R. Abstract data types and software validation. Communications of the ACM, New York, 21(12):1048-64, Dec. 1978.

12. _____. The design of data type specifications. In:YEH, R.T; Current trends in programming methodology. Englewood Cliffs, Prentice Hall, 1978. v.4, cap.4, p.61-79.

13. HILFINGER, P.N. et alii. An Informal definition of Alphard (preliminary). Pittsburg, Carnegie Mellon University, Department of Computer Science, Feb, 1978.

14. HOARE, C.A.R. Proof of correctness of data representations. Acta Informatica, Berlin, 1:271-81, 1972.

15. _____ & LAUER, P.E. Consistent and Complementary formal theories of the semantics of programming languages. Acta Informatica, Berlin, 3:135-53, 1974.

16. HOROWITZ, E. & SAHNI, S. Fundamentals of data structures. Maryland, Computer Science Press, 1976.

17. JONES, C.B. Software Development; a rigorous approach. London, Prentice - Hall International, 1980.

18. KOWALTOWSKI, T. Data structures and correctness of programs. Journal of the Association for Computing Machinery, New York, 26(2):283-301, Apr. 1979.

19. LEDGARD, H.F. & TAYLOR, R.W. Two views of data abstraction. Communications of the ACM, New York, 20(8): 382-4, June 1977.

20. LISKOV, B. et alii. Abstraction mechanisms in CLU. Communications of the ACM, New York, 20(8):564-76, Aug. 1977.

21. _____ & ZILLES, S. An introduction to formal specifications of data abstractions. In:YEH, R.T. Current trends in programming methodology. Englewood Cliffs, Prentice Hall, 1977. v.1, cap. 1, p. 1-32.

22. MAJSTER, M.E. Extendend directed graphs, a formalism for structured data and data structures. Acta Informatica, Berlin, 8(1):37-59, 1977.

23. _____. Limits of the algebraic specification of abstract data types. SIGPLAN Notices, New York, 12(10): 37-47, Oct. 1977.

24. PARNAS, D.L. A technique for software module specification with examples. Communications of the ACM, New York, 15 (5):330-6, May 1972.

25. _____ & HANDZEL, G. More on specification techniques for software modules. T.H. Darsmstadt, Fachbereich Informatik, 1975. (FG Betriebssysteme.1)

26. SHAW, Mary. The impact of abstraction concerns on modern programming languages. Pittsburgh, Carnegie - Mellon University, Department of Computer Science, Apr. 1980.

27. _____ & WULF, W.A. Abstraction and verification in Alphard: defining and specifying iteration and generators. Communications of the ACM, New York, 20(4):553-63, Apr. 1977.

28. SHNEIDERMAN, B. & SCHEUERMANN, P. Structured data structures. Communications of the ACM, New York, 17(10):566-74, Oct. 1974.

29. SMITH, J.M. & SMITH, D.C.P. Database abstractions:Aggregation. Communications of the ACM, New York, 20(6):405-13, June 1977.

30. STANDISH, T.A. Data structures - an axiomatic approach. In: YEH, R.T. Current trends in programming methodology. Englewood Cliffs, Prentice Hall, 1978. v. 4, cap. 3, p. 30-59.

31. TOSCANI, L.V. Técnicas de Verificação da Correção de Programas. Porto Alegre, PGCC da UFRGS, 1977.

32. WEGBREIT, B. & SPITZEN, J.M. Proving Properties of complex data structures. Journal of the Association for Computing Machinery, New York, 23(2):389-96, Apr. 1976.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Pós-Graduação em Ciência da Computação da UFRGS

Definição Formal de Tipos Abstratos
de Dados Através de um Exemplo

DISSERTAÇÃO APRESENTADA AOS SRS.

Raira Vieira Toscani

Visto e permitida a impressão

Porto Alegre, 12/06/81.

Coordenador do Curso de Pós-Graduação em
Ciência da Computação