PUC

Monografias em Ciência da Computação
nº 13/92

Uma Extensão de uma Linguagem para Bancos de Dados Funcionais LBF+

José de Jesús Pérez Alcázar
Mário Lacerda
Rosana S. G. Lanzelotte
Rubens N. Melo

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Monografias em Ciência da Computação, Nº 13/92
Editor: Carlos J. P. Lucena

Abril, 1992

Uma Extensão de uma Linguagem para Bancos de Dados Funcionais LBF+ *

José de Jesús Pérez Alcázar
Mário Lacerda
Rosana S. G. Lanzelotte
Rubens N. Mello

* Este trabalho foi patrocinado pela Secretaria de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio - Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
Brasil

Tel.:(021)529-9386 Telex:31078 Fax:(021)511-5645
E-mail:rosane@inf.puc-rio.br

RESUMO

Este relatório apresenta uma linguagem de programação de banco de dados baseada na linguagem DAPLEX. Ela é uma extensão da linguagem para banco de dados funcionais chamada LBF, com algumas das facilidades de orientação a objetos definidas por Atkinson, et ali.. Neste relatório também são destacadas as vantagens do modelo funcional e especificamente da proposta de Shipman e são analisadas algumas das extensões do modelo funcional para aplicações não-convencionais. A linguagem LBF+ mistura facilidades de outros sistemas como IRIS, PDM e O2. A idéia desta proposta é apresentar uma linguagem que de forma uniforme e concisa, permita ao usuário criar, operar, e atualizar o seu banco de dados e seus programas de aplicação.

PALAVRAS-CHAVE

Modelos de Dados Semânticos, Modelos de Dados Funcionais, Sistemas de Banco de Dados Orientados a Objetos, Linduagens de Programação de banco de Dados.

ABSTRACT

This report presents a data base programming language based on the DAPLEX language. It is an extension of the functional data base language called LBF, with some characteristics of the object-oriented database systems defined by Atkinson et ali.. In this report we also point out the advantages of the functional model and, specifically, the Shipman proposal and some of the function model extensions to non-standard aplications are revised. The LBF+ language mixes characteristics from other systems such as IRIS, PDM e O2. The idea of this proposal is to show a language which allows the user, in a concise and uniform way, to criate, operate and update his/her database and application programs.

KEYWORDS

Semantic Data Models, Functional Data Models, Object-Oriented Data Base Systems, Data Base Programming Languages.

SUMARIO

UMA EXTENSÃO DE UMA LINGUAGEM PARA BANCO DE DADOS FUNCIONAIS LBF+

1. INTRODUÇÃO.

Inicialmente, nos anos 60 até o início dos anos 80, os modelos de dados eram orientados para a representação de aplicações comerciais. O produto de sua utilização era chamado às vezes **esquema da empresa** [TsLo82] e representava a visão total de uma empresa, incluindo aplicações de contabilidade, folhas de pagamento, etc. Os modelos de dados neste período eram classificados em dois tipos: modelos clássicos e modelos semânticos. Os modelos clássicos (hierárquico, rede e realacional), inicialmente definidos, são orientados a registros [Kent79], portanto, uma série de limitações de modelagem são encontradas. Por esta razão os modelos semânticos foram criados. Eles nasceram da procura de modelos semanticamente poderosos que pudessem remover várias das restrições impostas pelos chamados modelos clássicos, o que permitia descrever a estrutura de um banco de dados de forma mais clara e precisa [SmSm77,HaMc81].

Entre os modelos semânticos, destaca-se pela sua simplicidade (e uniformidade), o modelo funcional [BuFr79, KePa76, Ship81]. Das várias propostas para o modelo funcional, algumas incorporam linguagens de manipulação de dados usando funções e conjuntos de operadores. Destas, somente as propostas de Shipman [Ship81] e Buneman [BuFr79] integram operações de manipulação de dados com operações de propósito geral em uma só linguagem, o que facilita a modelagem do comportamento das aplicações. Isto representa uma vantagem desta proposta em relação aos outros modelos semânticos, já que a maioria destes modelos são bons na modelagem de estruturas complexas (generalização, agregação, classificação e associação [Brod83]) mas são restritos na modelagem do comportamento.

Com a utilização de banco de dados em aplicações não-convencionais (CAD, CAM, CASE, Automação de Escritórios, etc [Melo88]), cujos domínios são complexos e altamente dinâmicos, os modelos clássicos e semânticos demonstraram não ser muito adequados. No caso dos modelos semânticos, eles tém-se baseado mais na construção de dados complexos através de mecanismos como atributos, agregação e generalização, que são mecanismos adequados para aplicações comerciais. Para a modelagem das novas aplicações que são geralmente interativas e altamente dinâmicas são necessários, além dos anteriores, novos mecanismos ou características.

Existe na literatura várias abordagens para tratar o problema de utilização dos bancos de dados não convencionais, porém não existe, um consenso do que seria um modelo ou SGBD para estas aplicações [SRLG90, ABDD89]. Existe entretanto um consenso sobre algumas das características que um modelo de dados deve preencher. Entre estas características podemos enumerar: facilidades para manipulação e definição de objetos complexos,

identidade de objetos, encapsulamento, herança (generalização), facilidades para a definição de tipos e/ou classes, completude computacional, etc.

No que diz respeito à completude computacional existem alguns problemas que precisam ser solucionados. Existe a crença de que o principal gargalo para a produtividade do programador de aplicações é o "descasamento de impedância" entre o banco de dados e as linguagens de programação [LeRi89]. Seguindo esta crença ,nós projetamos um sistema que mistura a tecnologia de banco de dados e a tecnologia de linguagens de programação para construir um sistema completo, com as facilidades de um SGBD e de uma linguagem de programação. Este tipo de ferramenta é chamada de Linguagem de Programação de Banco de Dados (DBPL) [AtBu87]. A idéia nasceu da necessidade de integrar linguagens de programação com persistência, importante quando se faz programação de ponto grande. Desta maneira a persistência tem sido incorporada às linguagens de programação de propósito geral e mais recentemente às linguagens orientadas a objetos [Khos89]. O PS-Algol [ABCC83] é o exemplo clássico de uma linguagem onde a persistência é ortogonal ao tipo, isto é, qualquer tipo de dado pode ser persistente. Os objetos persistentes podem ser mapeados e manipulados por um gerenciador de objetos. Outros exemplos de linguagens de programação persistente são Amber [Card85], que é uma linguagem de programação tipada, Galileo [ALCO85], que é também uma linguagem fortemente tipada com hierarquia de tipos e mais recentemente O2FDL [MaCB90], que é uma linguagem interativa fortemente tipada que integra os paradigmas de orientação a objetos com o funcional.

Nossa proposta é estender o modelo funcional de Shipman [Ship81] com algumas das facilidades de orientação a objetos como: construtores de objetos (tupla, lista, etc.), encapsulamento e extensibilidade para que desta maneira o usuário possa definir funções/operações computacionalmente complexas.

Esta nova linguagem chamada LBF+ (Linguagem para Banco de Dados Funcionais, Estendida), é uma linguagem de programação de dados, interativa que combina facilidades de linguagens utilizadas por sistemas como o IRIS [WILH90], o PROBE (PDM) [MaDa86], OODAPLEX [Daya89], e O2 [LeRi89].

Este trabalho tem as seguintes motivações:

- prover o ambiente EITIS [Melo87] com uma LPBD poderosa para desenvolvimento de aplicações não convencionais;

- aproveitar o conhecimento obtido na implementação de uma linguagem para bancos de dados funcionais, LBF, baseada em DAPLEX e que foi desenvolvida na UFMG [Pere88].

- estender a proposta de Shipman que apesar de ser bastante interessante carece de muitas facilidades necessárias para a modelagem de aplicações não-convencionais [KeWa87].

- investigar como obter essa extensão através de funções, de maneira uniforme e sem perder a simplicidade do modelo original.

Este trabalho será dividido em quatro seções, na seção 2, nós descreveremos a proposta funcional e especificamente a linguagem DAPLEX, suas vantagens e suas limitações. Na seção 3, descreveremos a proposta de orientação a objetos e alguns dos modelos funcionais também considerados como orientados a objetos. Na seção 4, descreveremos o nosso modelo proposto e a linguagem LBF estendida. Por último, na seção 5, apresentaremos algumas conclusões e propostas para pesquisas futuras.

2. MODELO FUNCIONAL

2.1 Introdução

O modelo funcional nasceu da busca de novas formas de modelagem generalizadas que pudessem ser utilizadas como base comum para a descrição e comparação dos três modelos clássicos [SIKe76]. Este formalismo deveria fornecer uma base unificada para o projeto de esquemas relacionais e de rede que fossem livres de anomalias de atualização. Uma outra utilização seria nos sistemas de bancos de dados heterogêneos, (centralizados ou distribuídos), onde o modelo funcional serviria como um mecanismo para a integração de diferentes esquemas (ex: MULTIBASE [Smit81]).

Esta seção apresenta o modelo funcional, descrevendo uma de suas principais propostas, o modelo de Shipman [Ship81].

2.2. Descrição do Modelo Funcional

O modelo funcional tem suas bases no conceito matemático de função e utiliza dois tipos de primitivas (estruturais) para a modelagem do mundo real [Kulk83]: os conjuntos de objetos e as funções que são aplicadas a estes para relacioná-los entre si. Os conjuntos de objetos estão divididos em conjuntos de entidades e conjuntos de valores. A principal diferença entre um conjunto de entidades e um conjunto de valores é que este último nunca faz parte do domínio de uma função. Os conjunto de valores são chamados por Kulkarni [Kulk83] conjuntos de entidades pré-definidas (ex: conjuntos de inteiros, reais, etc). As funções podem ser totais ou parciais, e a distinção, entre elas, corresponde a uma restrição [KaWo83] sobre o número de elementos do conjunto (domínio) que devem participar do relacionamento funcional.

O modelo funcional é classificado às vezes como um modelo do tipo irredutível [Date83, Brod84]. Isso porque uma função (fato atômico) é o único meio de relacionar um objeto a seus atributos (ou propriedades).

A fig. 2.1 apresenta a descrição gráfica, de acordo com o modelo funcional, do esquema conceitual de um banco de dados de uma biblioteca pessoal de artigos publicados em periódicos.

Existem algumas propostas para o modelo funcional e entre elas podemos destacar o modelo funcional de Kerschberg & Pacheco [KePa76], o modelo de dependências funcionais de Housel & Yao [YaWH82], o modelo de dados funcional de Katz & Wong [KaWo83], o modelo funcional de Buneman & Frankel [BuFr79] e o modelo funcional de Shipman [Ship81].

De todos estes modelos o mais destacado é o modelo proposto por Shipman, o qual está embutido na linguagem DAPLEX. Seu objetivo principal é fornecer uma linguagem conceitualmente

natural que sirva de interface para a utilização de um banco de dados. Mais na frente descreveremos rapidamente a linguagem DAPLEX.

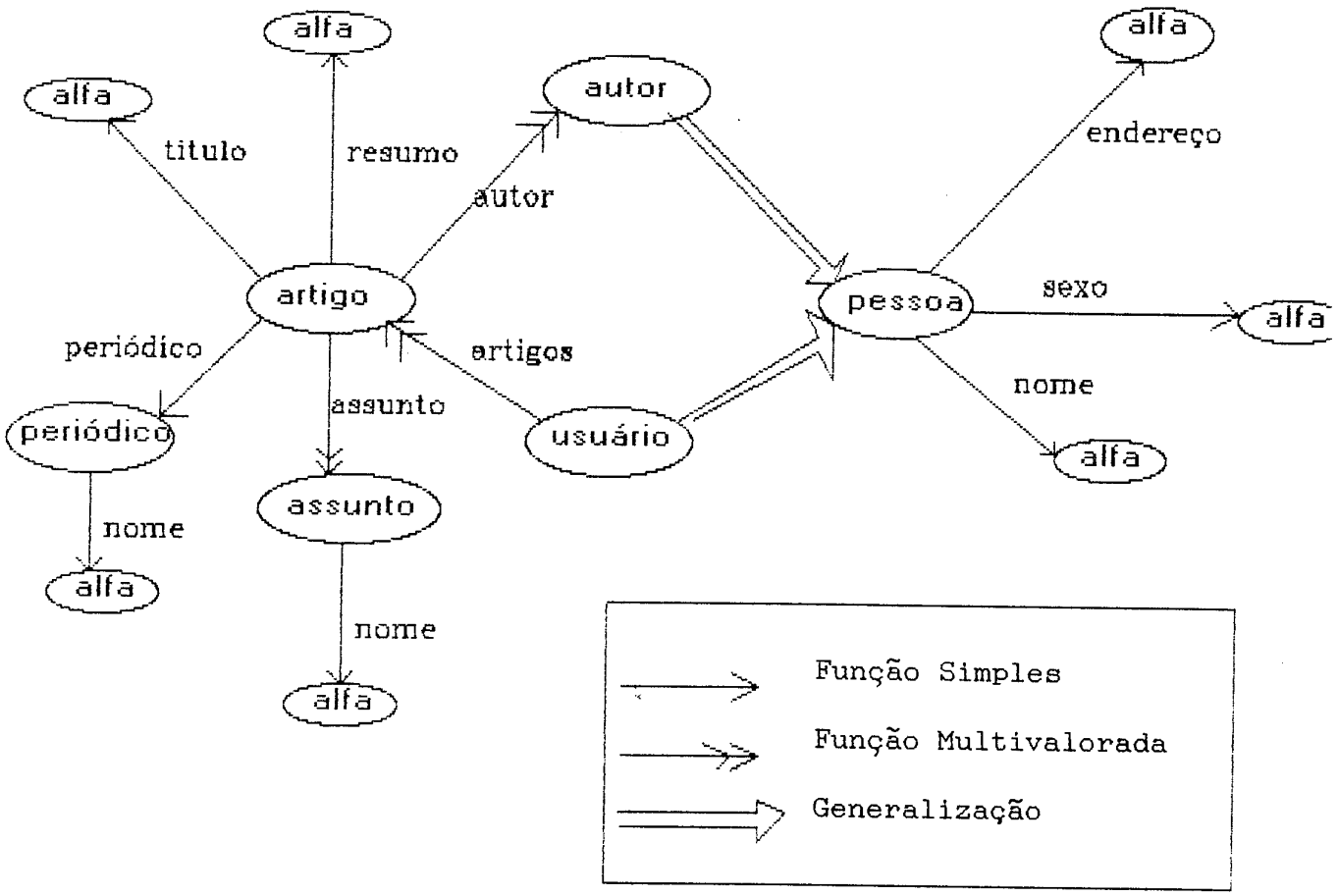

Fig. 2.1 Descrição gráfica do esquema conceitual (visão funcional)

2.3. Uma Avaliação dos Modelos Funcionais

Mas por que os modelos funcionais são uma ferramenta atraente para a modelagem conceitual? Kerschberg & Pacheco [KePa76], Kulkarni [Kulk83] e Orman [Orma84, Orma85] enumeram algumas destas razões:

- Os modelos funcionais fornecem um ambiente de modelagem semanticamente rico. O fato de não distinguir entre dados e programas (dados derivados) facilita bastante a modelagem

conceitual.

- As linguagens de consulta podem ser especificadas de forma mais natural. Além disso, facilidades de manipulação de dados podem ser integradas naturalmente com operações de propósito geral (FQL, a linguagem definida por Buneman & Frankel [BuFr79], é um exemplo).

- Existem algoritmos para mapear a estrutura de um banco de dados de rede (CODASYL) ou relacional para o modelo funcional [KePa76] e vice-versa. Portanto, o modelo de dados funcional pode ser usado como mecanismo para a integração de bancos de dados heterogêneos [Smit81].

- O modelo funcional representa a informação através de fatos atômicos (funções) [Orma85], eliminando alguns problemas de redundância e evitando anomalias de atualização.

- A teoria matemática de funções pode ser explorada para fornecer uma base teórica sólida para bancos de dados.

- A função, segundo Orman [Orma85], é uma estrutura mais primitiva e poderosa que outras comumente usadas pelos modelos já conhecidos, como relações, arquivos, conjuntos DBTG e arranjos. A partir de funções, consultas e restrições são naturalmente implementadas, já que consultas são funções aplicadas a valores de entrada e que retornam valores de saída, enquanto restrições são predicados que correspondem a funções cujo contra-domínio é o conjunto de valores lógicos.

- Funções podem ser consideradas caminhos lógicos definidos através do esquema de um banco de dados. É possível obter implementações eficientes de modelos funcionais por meio da otimização desses caminhos.

- Os modelos funcionais permitem a autodescrição de um banco de dados através de um simples metaesquema o qual também pode ser representado e manipulado com as mesmas construções do modelo, isto é, sem a necessidade de construções adicionais [Orma84].

2.4. A Linguagem DAPLEX

Por causa de sua grande influência na nossa linguagem proposta, nesta seção faremos uma breve descrição da linguagem DAPLEX. Faremos também uma avaliação desta proposta. Uma descrição detalhada da linguagem pode ser encontrada em [Ship81] ou [Pere88].

2.4.1. Estruturas

O esquema de um banco de dados funcional consiste de uma ou mais declarações ou definições de funções. A fig 2.2 ilustra a definição de um esquema funcional para o

banco de dados do exemplo descrito na Fig 2.1 (esta descrição é similar a uma rede semântica [Qui68]).

Entre as características principais da linguagem DAPLEX podemos enumerar as seguintes:

- Funções podem ter zero ou mais argumentos. As funções sem argumentos como: ARTIGO() ou PESSOA(), representam tipos de entidade e as funções com um ou mais argumentos representam relacionamentos ou atributos. Por exemplo: TITULO(ARTIGO) que representa o atributo TÍTULO do tipo ARTIGO, ou a função PERIODICO(ARTIGO) que representa o relacionamento entre PERIÓDICOS e ARTIGOS (em um só sentido). O conceito de tipo utilizado em DAPLEX é igual ao conceito de classe encontrada na literatura [ABBD89, Beer90].

```
DECLARE ARTIGO ()  ->> ENTITY
DECLARE PESSOA ()  ->> ENTITY
DECLARE USUÁRIO ()  ->> PESSOA
DECLARE AUTOR ()  ->> PESSOA
DECLARE ASSUNTO ()  ->> ENTITY
DECLARE PERIÓDICO ()  ->> ENTITY
DECLARE TITULO (ARTIGO)  -> STRING
DECLARE RESUMO (ARTIGO)  -> STRING
DECLARE PERIÓDICO (ARTIGO)  ->> PERIÓDICO
DECLARE AUTOR (ARTIGO) ->> AUTOR
DECLARE ASSUNTO (ARTIGO)  ->> ASSUNTO
DECLARE NOME (PESSOA)  -> STRING
DECLARE ENDEREÇO (PESSOA)  -> STRING
DECLARE SEXO (PESSOA)  -> STRING
DECLARE NOME (PERIÓDICO)  -> STRING
DECLARE NOME (ASSUNTO)  -> STRING
DECLARE ARTIGOS (USUÁRIO)  ->> ARTIGO

DEFINE ARTIGOS(AUTOR) ->> INVERSE OF AUTOR(ARTIGO)
DEFINE ASSUNTO(AUTOR) ->> ASSUNTO(ARTIGOS(AUTOR))
DEFINE ARTIGOS(ASSUNTO) ->> INVERSE OF ASSUNTO(ARTIGO)
DEFINE ARTIGOS(PERIODICO) ->> INVERSE OF PERIODICO(ARTIGO)
DEFINE ASSUNTO(PERIODICO) ->> ASSUNTO(ARTIGOS(PERIODICO))
DEFINE USUARIO(ARTIGO) ->> INVERSE OF ARTIGOS(USUARIO)
```

Fig. 2.2 Definição de um esquema funcional em DAPLEX

- Funções podem ser classificadas em: simples (retornam uma entidade) e multivaloradas (retornam um conjunto de entidades). Isto é representado em DAPLEX através de uma seta simples ou uma seta dupla. A primeira indica que a função declarada é simples como no caso da função NOME(AUTOR), a segunda indica que a função declarada é multivalorada como no caso da função ARTIGOS(AUTOR).

- Existe uma distinção entre uma entidade e sua identificação externa ou chave (identidade de objetos [KhCo86]).

- Entidades podem ser organizadas em uma hierarquia de tipos

onde atributos e relacionamentos são automaticamente herdados dos supertipos por cada um dos subtipos. Todo tipo de entidade definido em um banco de dados é um subtipo do tipo ENTITY. Desta maneira, ENTITY está no topo de uma hierarquia de tipos que pode ser estendida a qualquer nível. Por exemplo, na descrição da fig 2.2, o tipo de entidade PESSOA tem como subtipos os tipos de entidade AUTOR e USUÁRIO.

Como consequência, uma instância de AUTOR ou de USUÁRIO é também uma instância de PESSOA. Em geral, uma instância de um subtipo será uma instância de todos os seus supertipos e herdará todas as propriedades definidas sobre eles. Por exemplo, um AUTOR, pelo fato de ser uma PESSOA, terá como propriedades um NOME e um ENDEREÇO. Um grafo representando a hierarquía de tipos definida no esquema funcional da Fig 2.2 é mostrado na Fig 2.3.

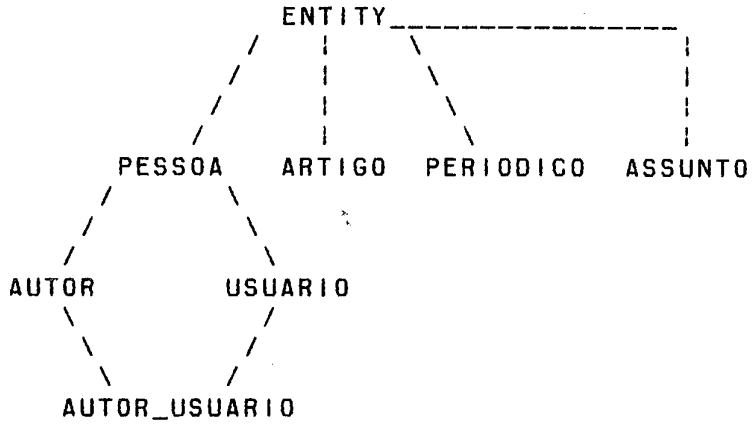

Fig 2.3 Exemplo de Hierarquia de Generalização

- A partir de funções básicas é possível criar funções derivadas, isto é, o conceito de dados derivados [KoPa81] é suportado de forma simples e natural. Funções derivadas são definidas usando o comando DEFINE. Por exemplo, na fig. 2.2, a função ASSUNTO(AUTOR) que representa os assuntos nos quais um determinado autor publica, é definida utilizando a composição das funções ASSUNTO(ARTIGO) e ARTIGOS(AUTOR).
  A partir de dados derivados o modelo permite a definição de visões do usuário. As consultas (ou programas) podem ser facilmente projetadas através de uma progressiva definição de funções derivadas. Além disso, as funções derivadas são mecanismos importantes para a captura da semântica na descrição dos dados.

- Já que o modelo funcional representa a realidade através de fatos atômicos, funções, é muito fácil chegar a um esquema conceitual sem a necessidade de tratar problemas de normalização presentes nos modelos orientados a registros. Além disso, estas funções podem ser naturalmente acrescentadas ao esquema o que permite a sua fácil evolução e o projeto incremental do banco de dados.

- Este modelo permite um conjunto completo de operacões que pode ser naturalmente acrescido de operações de proposito geral. A formulação de consultas é feita de uma maneira bem natural, livre de aspectos de implementação, isto é, o usuário não precisa ter em conta aspectos físicos já que a forma como as funções são implementadas é totalmente transparente.

2.4.2. Operações

Embora DAPLEX integre operações de definição e manipulacão de dados, ela foi definida por Shipman para ser embutida numa linguagem de alto nível. Portanto, operações de proposito geral (como operações aritméticas) e operações de E/S não são definidas.

DAPLEX utiliza expressões e comandos como construções básicas para a manipulação dos dados. As expressões sempre aparecem dentro de comandos e representam tanto um conjunto de entidades quanto uma só entidade, o qual permite classificá-las em expressões multivaloradas e simples.

Os comandos dirigem o sistema para desenvolver uma ação. Estes comandos podem ser de definição de dados (já estudados) ou de manipulação. Os comandos de manipulação podem ser divididos em comandos de controle e comandos de atualização. DAPLEX utiliza um só tipo de comando de controle, o comando FOR. Um exemplo da utilização deste comando de iteração é o seguinte:

```
FOR EACH PESSOA
    PRINT NOME(PESSOA)
```

Neste exemplo são listados os nomes de todas as pessoas registradas no banco de dados. O comando FOR, neste caso, itera sobre o conjunto de entidades do tipo PESSOA e executa o comando PRINT sobre cada membro do conjunto. Aqui pode ser visto que os tipos são coleções de entidades (objetos). Outros modelos tratam os tipos como um protocolo de especificação (ex: Vbase [AnHa87]). Não existe diferença explicita entre tipos e classes, esta característica é herdada pelos modelos baseados em DAPLEX, como PDM [MaDa86], IRIS [WiLH90], etc. O comando FOR pode ser aplicado sobre uma expressão multivalorada ou sobre uma expressão simples. Um exemplo do último caso é:

```
FOR THE AUTOR SUCH THAT
        NOME(AUTOR) = "Tsichritzis"
    PRINT ENDERECO(AUTOR).
```

2.4.3. Uma Avaliação da Proposta de Shipman

A pesar de todas as suas vantagens a proposta de Shipman padece das seguintes falhas:

- O fato das funções simples serem definidas como totais, resulta em duas desvantagens:

  . Não é possível introduzir no banco de dados entidades cuja informação é incompleta (algumas propriedades não são conhecidas).

  . As funções inversas não podem ser definidas em todos os casos.

- Shipman propõe embutir DAPLEX numa linguagem de proposito geral o que gera "mismatch impedance" com todos seus problemas associados [ZdMa89].

- Além do anterior, entre outras coisas, DAPLEX carece de:

  . Operadores para eliminar funções e tipos de entidades do banco de dados.

  . Mecanismos para a inclusão e exclusão de uma entidade da extensão de um tipo especificado.

  . Facilidades para a definição de funcões recursivas que constituem um mecanismo poderoso para modelagem de dados (consultas mais complexas podem ser definidas).

Trabalhos para solucionar estes problemas foram desenvolvidos em EFDM [Kulk83] e LBF [Pere88]. Os dois sistemas estendem a linguagem DAPLEX com operadores de propósito geral gerando assim linguagens auto-contidas, isto é, linguagens persistentes usadas como ferramentas para a implementacáo de banco de dados. Entretanto, estas extensões mantém algumas das falhas de DAPLEX no que diz respeito à modelagem de aplicações definidas sobre domínios complexos, como: CAD, CAM, CASE, etc. Kemper & Walrath [KeWa87] enumeram algumas críticas da utilizacão de DAPLEX a aplicações em engenharia:

- Não permite ao usuario definir funcões computacionalmente complexas. Para definir manipulacões de objetos complexos, DAPLEX teria que permitir a definição de operacões complexas por parte do usuário. Isto e, precisa-se de um mecanismo de definição de Tipos Abstratos de Dados (TADs).

- Inserir e recuperar objetos complexos (isto é, objetos que são compostos de outros objetos) em DAPLEX é extremamente custoso, já que ela representa estes objetos através de composição de funções (aninhamento de funcões) o que implica que muitas funções têm que ser definidas para fornecer construções que facilitem a manipulacão dos dados. Uma solução para este problema, vamos ver mais na frente, será a definição de construtores como: lista e tupla (conjunto, já faz parte do modelo funcional de Shipman).

3. MODELOS DE DADOS PARA APLICAÇÕES NÃO-CONVENCIONAIS

3.1 Introdução

Vários domínios, de aplicação, incluindo CAD, CAM, CASE, Automação de escritórios e bases de conhecimentos, precisam da representação direta e a manipulação eficiente de objetos complexos arbitrários. Os objetos complexos são tipicamente construídos com agregadores (ex: tuplas) e coleções de objetos aninhados recursivamente. Estes domínios de aplicação diferentes, impõem diferentes tamanhos, acessos e características estruturais sobre seus objetos.

Além disto, como foi mencionado anteriormente, estes domínios são altamente dinâmicos, implicando na existência de novos mecanismos mais ricos para a modelagem do comportamento. Estes mecanismos, possivelmente herdados das linguagens de programação, são mais orientadas para a modelagem do comportamento que os modelos de dados.

O campo de banco de dados orientado a objetos tem sido originado da convergência de várias linhas de pesquisa. Os campos de linguagens de programação, inteligência artificial (modelos semânticos) e engenharia de software, têm contribuido para o uso da tecnologia de orientação a objetos na área de banco de dados [ZdMa89].

Existe uma atividade experimental muito forte na área de banco de dados orientado a objetos. Entretanto, não existe uma definição comum de um modelo orientado a objetos e consequentemente não existe uma base formal que o suporte. Na realidade o que existe é um consenso sobre as características que deve possuir um sistema de banco de dados orientado a objetos. Estas características são mencionadas em [ABDD89]. Eles dividem estas características em obrigatorias, opcionais e em aberto.

As características obrigatórias, são aquelas que o sistema deve satisfazer para ser declarado um sistema de banco de dados orientado a objetos. Isto é, ele deve ser um Sistema de Banco de Dados (SBD) e um Sistema Orientado a Objetos (SOO). Como SBD ele deve suportar:

- **Persistência.** Os dados devem continuar existindo apos a execução dos programas;
- **Manipulação de Memória Secundária;**
- **Concorrência;**
- **Recuperação;** e
- **Definição de consultas "ad hoc".** O que implica que o sistema deve suportar interfaces declarativas, eficientes e independentes da aplicação.

Como SOO ele deve suportar:

- **Objetos Complexos**;

- **Identidade de Objetos**;

- **Herança**;

- **Encapsulamento.** Esta idéia tem sua origem na engennaria de software e vem da necessidade de distinguir claramente entre a especificação e implementação de uma operação (ou objeto) e da necessidade de modularizar. Entretanto ha casos onde o encapsulamento não é necessário. Por exemplo, no caso de consultas "ad hoc";

- **Polimorfismo.** O sistema deve permitir que alguns valores e variáveis possam ter mais de um tipo. Isto permite uma maior flexibilidade, embora dificulte um pouco a checagem de tipos. O polimorfismo foi classificado por Cardelli & Wegner [CaWe85] como:

  1) Universal. Quando uma função executa o mesmo codigo para argumentos de um numero infinito de tipos, que apresentam uma estrutura em comum. Ele subdivide-se em:

     a) Paramétrico. Uma função pode trabalhar para muitos tipos, executando geralmente o mesmo trabalho independentemente dos argumentos. Por exemplo, o mecanismo "generic" de Ada [Barn83].

     b) Inclusão. Usado em caso de subtipos, isto e, uma função definida para um tipo pode ser aplicada a seus subtipos.

  2) Ad-hoc. Quando uma função trabalha, sobre diferentes tipos (número finito), que podem não ter uma estrutura comum, podendo executar códigos diferentes para cada tipo de argumento. Ele subdivide-se em:

     a) "Overloading" (Sobrecarga). O mesmo nome é usado pra denotar diferentes funções e o contexto é usado para decidir qual delas é usada.

     b) "Coercion" (Coação). E uma abstracão sintatica. Converte o argumento para o tipo esperado pela funcão.

- **Completude Computacional.** O sistema deve permitir a definição de uma aplicação completa sem precisar da importacão de outras linguagens ou facilidades de outros sistemas, o que evita o problema de "mismatch impedance".

- **Tipos e Classes** Alguns sistemas têm unicamente têm unicamente tipos (ex: FAD [DaKV88], outros têm unicamente classes (ex: ORION [BCGK87]) e nos dois casos estes

conceitos são sobrecarregados. Tradicionalmente, os tipos têm dois papéis: um tipo descreve uma estrutura, e tambem descreve uma extensão, o domínio dos elementos que têm esta estrutura. As classes tambem descrevem uma extensão, cada classe descreve uma colecão de objetos. A diferenca é que enquanto a extensão de um tipo e definida pela sua estrutura associada, a extensão de uma classe é definida pelo usuário. Sendo assim, Beeri [Beer90] considera estes dois conceitos necessários: classes, cujas instâncias são objetos e tipos, cujas instâncias são valores.

Em [ABDD89], também são relacionadas algumas características desejáveis mas que não são consideradas obrigatórias. Algumas características como por exemplo, suporte de versões, distribuição e projeto de transacões, são importantes para certo tipo de aplicações não-convencionais. Entretanto, existem outras que são utilizadas em várias aplicações mas não são muito importantes como: Herança múltipla (um tipo pode ser subtipo de mais de um tipo) e checagem e inferenca de tipos.

Algumas caracteristicas de grande importância na area de BDs, não foram consideradas em [ABDD89], como características obrigatórias. Tal fato ocorreu por falta de um consenso dentro dos autores. Estas características são mencionadas a seguir:

- Definição de visões e dados derivados;

- Facilidades para administracão de BDs;

- Restrições de integridade; e

- Facilidades de evolucão do esquema.

Analisando esta classificacão podemos perceber que não existe na comunidade de banco de dados um consenso sobre o que seriam as características básicas de um SGBDOO e mesmo de um SGBD para aplicações não convencionais [SRLG90]. Entretanto, nós achamos que estendendo o modelo funcional de dados com algumas das características definidas em [ABDD89] e pelo "Comittee for advanced DBMS Functions" [SRLG90], podemos obter um modelo uniforme, relativamente simples de usar que possa servir de base para SGBDs para aplicações não convencionais. Nossa crenca tambem é suportada pela existência de vários prototipos implementados em universidades e empresas que seguem esta abordagem. Como exemplo disto podemos enumerar os sistemas: IRIS [WiLH90] desenvolvido pela Hewlett-Packard, PDM [MaDa86] desenvolvido pela CCA, FUGUE [HeZd88], JASMIN [Mais88] desenvolvido pela Fujitsu, O2FDL [MaCB90] desenvolvido pela Universidade de Austin (TX), etc. Baseados no argumento acima citado, podemos dizer que existe uma certa aceitação desta abordagem dentro da comunidade científica. Nas seções seguintes descreveremos algumas das propostas na qual se baseia a LBF+.

3.2. Probe Data Model (PDM).

O sistema PROBE [DaSm86], é um sistema de gerenciamento de banco de dados extensível proposto pela Computer Corporation of America (agora, Xerox Advanced Information Technology), cujo modelo PDM [MaDa86, DMBC87] é baseado em DAPLEX. Ele tem como objetivo prover mecanismos gerais para atender as necessidades de aplicações não convencionais. Alguns destes mecanismos DAPLEX já fornecia, por exemplo: identidade de objetos, herança, etc.

As extensões feitas em DAPLEX tiveram como objetivos [MaDa86]:

- Incorporar no modelo, funções com varios argumentos e funções computadas (procedimentos gerais).

- Obter uma definição formal do modelo, baseada na álgebra (nos moldes do modelo relacional) para ser usada como base na otimização de consultas, mapeamento de visões e estudos para o desenvolvimento de linguagens de consulta.

- Prover uma forma "limpa" de extensão do modelo, com objetos tendo semântica não convencional, particularmente espacial e temporal.

A idéia do sistema PROBE é fornecer mecanismos gerais para o suporte de aplicações não-convencionais ao contrario de mecanismos (construtores) ad-hoc de propósito especial, como e o exemplo dos relacionamentos de composição de objetos (COMPONENT-OF definido em [Katz86]). Esta proposta de fornecer mecanismos gerais é também seguida pelo grupo POSTGRES [Ston86]. Entretanto, enquanto POSTGRES estende um SGBD relacional, PROBE inicia com um modelo de dados semântico que já fornece varias das características básicas para a modelagem de aplicacões não convencionais.

3.2.1. Estruturas.

O PDM utiliza dois tipos básicos de estruturas: as entidades e as funções. Com estes dois conceitos, ele consegue representar objetos complexos. Uma diferença entre o PDM e o DAPLEX neste sentido, é a definição explícita de tipos de entidade. Em DAPLEX um tipo é definido como uma função de zero argumentos. Entretanto, esta diferença é somente sintática.

Objetos em PDM, da mesma maneira que na linguagem DAPLEX, são adicionados explicitamente a tipos. Esta caracteristica e interessante pois normalmente nos SGBDOOs, os objetos são criados como pertencendo a algum tipo e suas propriedades e operacões são determinadas pelo tipo no qual foi criado. Em PDM, os objetos (entidades) podem entrar e sair dos tipos dinamicamente.

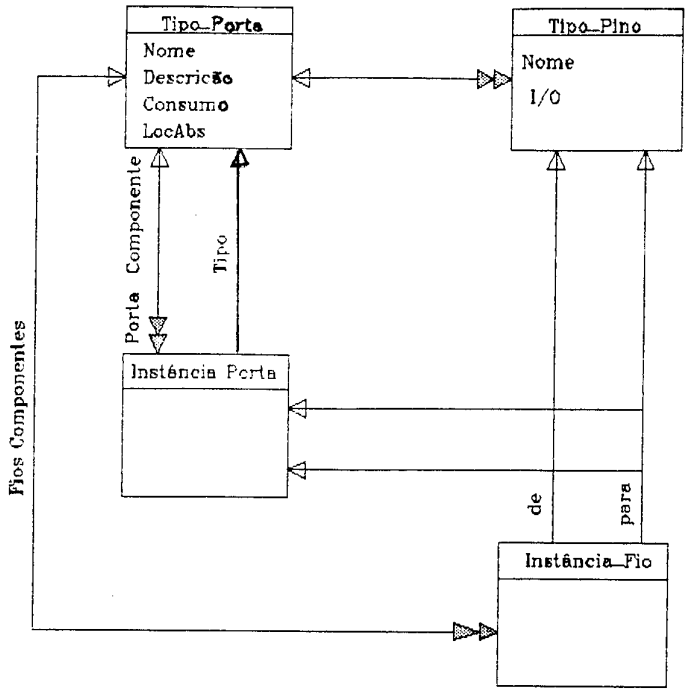

Fig. 3.1 Representação Gráfica de um objeto complexo.

Um objeto complexo e seus componentes são modelados como entidades; relacionamentos entre o objeto complexo e seus componentes, bem como entre componentes, são modelados como funções que retornam uma ou várias entidades (conjunto). Assim o esquema inclue tanto a interface quanto a implementação (usando a terminologia definida em [Bato84]) de um objeto complexo, junto com uma ou mais funções que representam os relacionamentos entre eles [DMBC87].

A fig. 3.1 mostra a representação gráfica de um objeto complexo, uma porta AND de 4 entradas, em PDM (exemplo extraído de [DMBC87]). A definição deste objeto na linguagem de definicão de dados do PDM pode ser achada na fig. 3.2.

As funções em PDM podem ser definidas "intencionalmente" (similar aos dados derivados em DAPLEX) ou definidos "extensionalmente" (suas extensões são armazenadas no BD). Conceitualmente, as funções definidas "extensionalmente" são tratadas como tabelas contendo uma tupla para cada combinação dos parâmetros de entrada ou saída. PDM suporta uma forma de ver uma tabela como uma família de funções. Por exemplo, uma tabela com três colunas pode ser tratada como uma função com uma entrada e duas saídas, ou duas entradas e uma saída, ou nenhuma entrada e três saídas ou três entradas e uma saída "booleana". Este uso é similar a um predicado em PROLOG ou as funções predicado do IRIS que veremos na seção 3.4.

```
Entity      TIPO_PORTA is ENTITY
Function    Nome(TIPO_PORTA) -> STRING
Function    Descricao(TIPO_PORTA) -> STRING
Function    Pinos(TIPO_PORTA) -> set of TIPO_PINO
Function    Consumoenergia(TIPO_PORTA) -> INTEGER
Function    LocAbs(TIPO_PORTA) -> POINT
Function    Portas_Componentes(tipo_porta) -> set of
                                              INSTANCIA_PORTA
Function    Fios_componentes(TIPO_ PORTA) -> set of
                                              INSTANCIA_FIO

Entity      TIPO_PINO is ENTITY
Function    Nome(TIPO_PINO) -> STRING
Function    I/O(TIPO_PINO) -> "I", "O"

Entity      INSTANCIA_PORTA is ENTITY
Function    Nome(INSTANCIA_PORTA) -> STRING
Function    Tipo(INSTANCIA_PORTA) -> TIPO_PORTA
Function    LocAbs(INSTANCIA PORTA) -> POINT
Function    LocRel(INSTANCIA PORTA) -> POINT

Entity      INSTANCIA_FIO is ENTITY
Function    De(INSTANCIA_FIO) -> (INSTANCIA_PORTA.
            TIPO_PINO)
Function    Para(INSTANCIA_FIO) -> INSTANCIA_PORTA.
            TIPO_PINO)
```

Fig. 3.2 Entidades e funções representando o objeto complexo.

```
for tp in TIPO_PORTA
   define
   Consumoenergia(tp) : =
               sum(Consumoenergia(tp2 in TIPO PORTA
                  Where tp2 isin
                        Tipo(Portas_componentes(tp))))

   for ip in INSTANCIA_PORTA
      define
      LocAbs(ip):= transformation(LocRel(ip).
                     LocAbs(tp in TIPO_PORTA
                     where ip isin Portas_componentes(tp)))
```

Fig. 3.3. Funções Derivadas.

As funções definidas intensionalmente são importantes para modelar valores de objetos complexos que são derivados de seus objetos componentes, bem como valores de componentes que podem ser determinados partindo-se de valores do objeto que o inclui. Por exemplo, em um circuito composto de outros circuitos, o consumo de energia deste é igual à soma dos consumos de energia de cada um de seus componentes. Ao contrário, a localizacão

absoluta do circuito, em uma tela gráfica, pode ser derivada recursivamente a partir de sua localização com relacão a seu pai cuja localização também é calculada com relacão a seu pai e assim sucessivamente. Sendo assim, no caso do exemplo do circuito da fig. 3.1, as funções CONSUMOENERGIA e LOCABS são calculadas pelos procedimentos mostrados na fig. 3.3. Este tipo de funcão e chamada de funcão derivada.

3.2.2. Operações

O PDM foi projetado a partir da experiência dos autores na definição de ADAPLEX [Smit83]. Portanto, a linguagem de definicão dos procedimentos e das consultas é muito semelhante a linguagem DAPLEX.

Além da linguagem anterior, no sistema PROBE foi desenvolvida uma álgebra (a álgebra PDM) que apresenta características que permitem a manipulação de objetos complexos:

a) Provê operações genéricas sobre entidades e funções;

b) Permite a definicão de operacões especificas da aplicacão;

c) Permite a especificacão de regras para propagar operacões ao longo dos relacionamentos.

Esta álgebra é uma modificacão da álgebra relacional e contém operações tais como selecão, projecão, produto cartesiano e outras operações sobre conjuntos. Para efeitos da álgebra, funções são vistas como relações, e tipos de entidades como funções sem argumentos (como DAPLEX). As operações são aplicadas a funções e têm como resultado, funções. Por exemplo:

SELECT (F, P): Seleciona aquelas tuplas da função F que satisfazem o predicado P. Os predicados são vistos como funcões booleanas.

PROJECT(F, L): Projeta as tuplas de F sobre as funções na lista L.

APPLY-APPEND(F, G): Modela a composição de funcões. É equivalente a operacão de juncão da álgebra relacional.

A álgebra também contém operacões de agregacão, atualizacão, e de metadados para criar novas funcões.

Esta álgebra não pretende ser uma linguagem de consulta para o usuário final; ela pode servir como base para a definicão da semântica de linguagens de consulta e tambem para facilitar a otimizacão das consultas [DMBC87].

Para permitir a inclusão de recursão no SGBD, a álgebra foi estendida com um operador de recursão, chamado "traversal recursion" que é uma generalização do fecho transitivo [RHDM86].

3.2.3. Conclusão

O modelo PDM é considerado na literatura como orientado a objetos. Entretanto, seu SGBD associado, não considera, ou considera com algumas limitações as características definidas como mandatórias em [ABDD89].

TIPOS E CLASSES.

Ele adota uma única declaração para tipo e extensão (classe), o que dificulta a definição de persistência. O anterior é importante porque no caso de completude computacional (ou definição de uma Linguagem de Programação de Banco de Dados, DBPL) não seria possível saber se o "extent" (extensão) é persistente ou não, principalmente na definição de valores temporários.

ENCAPSULAMENTO.

Como será visto na seção 3.4, nestes modelos baseados em DAPLEX, as entidades ou objetos não têm estrutura e não existe uma diferença muito clara entre atributo ou método. Em PDM não existe o conceito de encapsulamento do jeito que é definido na maioria dos modelos orientados a objetos. Porém a informação associada ao objeto só pode ser acessada através de funções. Existe no PDM um certo tipo de abstração, no sentido de separar a especificação (interface) de uma função, de sua implementação. Assim o tipo da função (computada ou armazenada) é transparente ao usuário.

OBJETOS COMPLEXOS.

A inexistência de alguns dos construtores básicos como por exemplo, Lista, e a falta de ortogonalidade na utilização dos outros construtores (tupla e consulta), dificultam a implementação de objetos complexos.

POLIMORFISMO.

Da mesma maneira que DAPLEX, o PDM permite dois tipos de polimorfismo: o de inclusão e de sobrecarga. Em DAPLEX quando uma função é aplicada a um objeto dado, ela é determinada pelo seu nome e pelo tipo do objeto para o qual ela é aplicada. Se a função for herdada e o tipo tiver vários supertipos com funções que têm esse mesmo nome (o nome da chamada), a função mais específica deve ser selecionada. Isto é, a função definida sobre o supertipo localizado mais perto dentro da árvore de tipos. Se esta não for encontrada (ambigüidade ou inexistência), um erro deve ser gerado. Estes conceitos também se aplicam a funções com vários argumentos. DAPLEX utiliza ainda a cláusula AS, que permite ao usuário mudar o "role" ou tipo de um objeto de acordo com a aplicação. Isto evita os problemas de conflitos de nomes de funções, no caso de herança múltipla. Os artigos sobre o PDM, [MaDa86] e [DMBC87], não descrevem nenhum mecanismo similar.

COMPLETUDE COMPUTACIONAL

Os artigos analisados não falam da existência de nenhuma linguagem hospedeira nem da existência de operacões de proposito geral dentro da LMD. Portanto, não podemos considerar este sistema computacionalmente completo.

Para terminar, podemos concluir que este modelo tem como vantagens a sua flexibilidade, a utilização de mecanismos para a manipulação de dados temporais e espaciais e a definição de uma álgebra que permite a otimizacão das consultas. Alem de tudo isto, suporta um mecanismo de recursão mais poderoso e a generalização do conceito de função de DAPLEX para permitir relacionamentos entre coleções de entidades e valores escalares. Por exemplo, a seguinte função,

QUANTIDADE(DEPOSITO) -> (MATERIAL, INTEGER)

pode ser declarada como função basica enquanto que em DAPLEX so seria possível como uma funcão derivada (utilizando a clausula COMPOUND).

3.3. IRIS

Este sistema está sendo desenvolvido nos laboratorios da Hewlett-Packard [WILH90, FACC89]. Um dos seus objetivos e ampliar a produtividade do programador de banco de dados, através de um modelo de dados expressivo. Outro objetivo é fornecer um suporte de banco de dados para o desenvolvimento e a integração de aplicações em áreas não convencionais.

A fig. 3.4, extraida de [WILH90], mostra a arquitetura do sistema. O núcleo ("Kernel") do Iris implementa o modelo de dados que é orientado a objetos e funcional. Este modelo é um modelo de dados semântico que suporta TAD's, e suas raízes podem ser achadas em DAPLEX e TAXIS [MyBW81]. Ele tem muitas similaridades com o PDM e entre outras coisas, pemite herança, propriedades gerais, restrições, operacões definidas pelo usuario, inferência, controle de versões e tipos extensíveis.

Como muitos outros SGBDs, o Iris e acessivel via interfaces interativas "stand-alone" ou como interfaces embutidas em linguagens de programação. No momento [WILH90], duas interfaces interativas são suportadas. Uma interface interativa, Object SQL (OSQL), que é uma extensão orientada a objetos do SQL. A outra e um sistema baseado em X-Windows que permite aos usuarios recuperar e atualizar valores das funções e metadados. Alem das interfaces anteriores, Iris suporta duas interfaces em "batch". A primeira, CLI ("C language interface"), que permite aos programadores acessar o Iris de uma maneira orientada a objetos. Este acesso é feito manipulando variáveis C que descrevem o banco de dados Iris, o metadados, e os objetos no banco de dados. A segunda interface consiste em embutir OSQL em várias

linguagens hospedeiras.

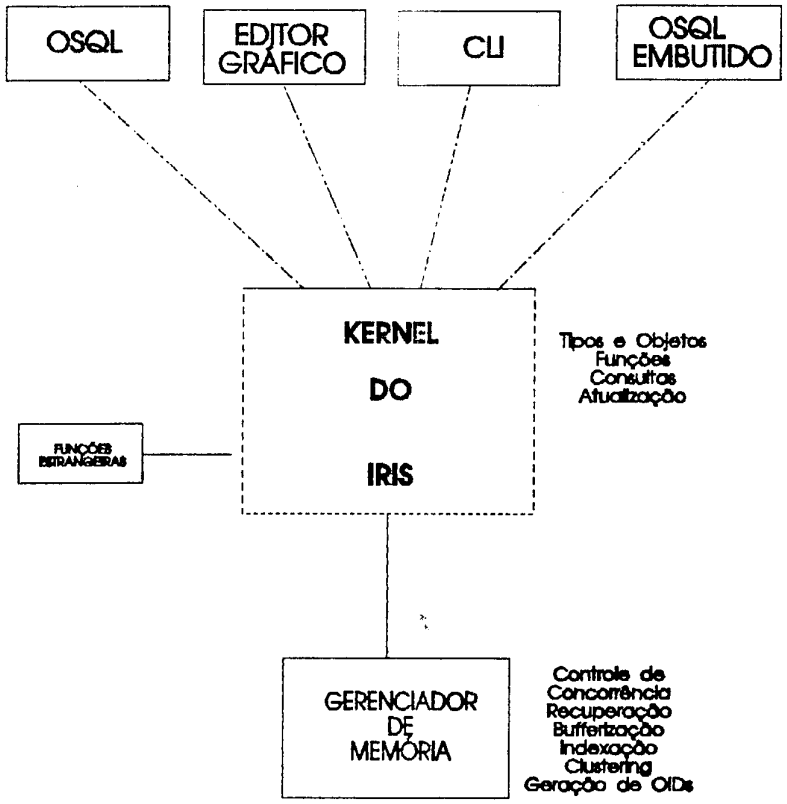

Fig. 3.4 Arquitetura do Sistema IRIS

Zdonik & Maier [ZdMa89], declaram que o Iris foi projetado para suportar persistência em múltiplas linguagens de programação enquanto que outros sistemas, por exemplo O2 [Deux90], foram projetados para suportar múltiplas linguagens de bancos de dados para escrever os métodos ou operações.

O modelo do Iris possui três construcões importantes: objetos, tipos e funções. Os objetos, os tipos e as funções definem a estrutura do banco de dados. Entretanto, as funções também são utilizadas para definir o comportamento dos objetos, isto é suas operações.

3.3.1. Objetos e Tipos.

Objetos no modelo Iris são itens atômicos, que podem servir como argumentos e resultados de operações. Cada objeto está associado com um ou mais tipos; estes tipos atuam como restrições, determinando em quais operações o objeto pode servir como argumento. Alguns objetos tais como inteiros, caracteres e listas, são chamados de literais e são representáveis diretamente. Cada tipo literal tem uma extensão ("extent") fixa. Os outros

objetos, não literais, são representados através de um "surrogate" ou identificador de objetos. Exemplos destes objetos, são: os tipos, as funções e os objetos abstratos (ex: estudantes e professores).

Os tipos são organizados em um grafo dirigido (acíclico) que suporta generalização e especialização (herança). Um tipo pode ter múltiplos subtipos imediatos e múltiplos supertipos imediatos. O tipo OBJECT é um tipo ancestral de todos os outros tipos e contém todos os objetos.

O modelo do Iris permite que o grafo de tipos mude dinamicamente. Sendo assim, os Tipos podem ser adicionados e removidos, e os objetos podem mudar seu tipo (flexibilidade similar ao PDM). Este tipo de facilidade é útil para modelar situações do mundo real, mas introduz problemas na checagem de tipos.

Do mesmo modo que o PDM, os tipos no Iris são coleções de objetos. Outros modelos tratam os tipos como um protocolo de especificação (ex: Vbase [AnHa87]). Dentro da hierarquia de tipos de Iris, é possível restringir um conjunto de sub-tipos para que sejam mutuamente exclusivos. Tal restrição não é uma característica típica dos tipos em linguagens de programação, por exemplo, Vbase e Gemstone [BMOP89] não suportam esta facilidade. Esta característica está mais ligada a linguagens para a representação do conhecimento.

3.3.2 Funções.

Todas as operações no Iris são funções. As propriedades dos objetos e seus relacionamentos entre objetos também são modelados em termos de funções, que são definidas sobre seus tipos. O Iris diferencia funções de recuperação de dados de funções de atualização de dados. As funções de recuperação não podem ter efeitos colaterais, o que facilita a depuração das aplicações.

As funções são especificadas em duas partes, declaração e implementação. A declaração especifica o nome da operação, número e tipo de parâmetros e os resultados. Por exemplo:

CREATE FUNCTION SUPERVISOR_DE(EMPREGADO) -> EMPREGADO **FORWARD**

cria uma função chamada SUPERVISOR_DE. A cláusula FORWARD declara que a implementação da função será especificada mais tarde.

O Iris permite vários tipos de implementação. A separação da declaração da função de sua implementação, facilita a abstração de dados porque permite que os usuários mudem dinamicamente a implementação. Sendo assim, os programas de aplicação não são afetados. Uma forma de implementar uma função consiste em armazená-la como uma tabela, que registra explicitamente os objetos de entrada e de saída. Esta maneira equivale às funções

armazenadas em PDM. Se SUPERVISOR_DE fosse armazenada, então a implementação seria especificada como:

```
CLUSTER SUPERVISOR_DE.
```

Se a cláusula FORWARD for omitida na declaracão, assume-se que a função é armazenada.

Outra forma de implementar uma função é como uma consulta, cujo resultado é uma tabela virtual (a função é especificada em termos de outras funções). A consulta é avaliada cada vez que a função é invocada (equivale à função derivada do PDM). Por exemplo, se SUPERVISOR_DE fosse uma função derivada, então a implementação seria especificada como:

```
DERIVE FUNCTION SUPERVISOR_DE(EMPREGADO E) -> EMPREGADO AS
                SELECT S FOR EACH EMPREGADO S
                WHERE S = CHEFE_DE(DEPARTAMENTO_DE(E))
```

Para melhorar o desempenho, Iris suporta funções derivadas materializadas. Isto, é similar à noção de visões materializadas em sistemas relacionais. Estas funções não são regeneradas automaticamente.

Por último, um outro tipo de implementação de funções, são as funções estrangeiras ("foreign functions"). Uma função estrangeira é escrita em alguma linguagem de programação tal como C ou LISP. Estas funções estão fora do controle do sistema e podem ter efeitos colaterais arbitrários. Se SUPERVISOR_DE fosse estrangeira, então a sua implementação seria especificada como:

```
LINK FUNCTION SUPERVISOR_DE TO "CalcSup.obj"
```

O arquivo "CalcSup.obj" deverá conter o código de máquina para implementar SUPERVISOR_DE.

O Iris utiliza três comandos para atualizar funções:

SET : Para atualizar funções simples ou monovaloradas;
ADD : Para adicionar valores a funções multivaloradas;
REMOVE : Para remover valores de funções multivaloradas.

3.3.3 Uma Interface SQL.

Para consultar o banco de dados, SQL foi estendida para adaptá-la aos paradigmas de modelagem funcional de dados e modelagem por objetos. Esta linguagem é chamada OSQL e permite referenciar os objetos diretamente e utilizar funções definidas pelo usuário e pelo sistema dentro das clausulas SELECT e WHERE. Ela é semelhante (mais restrita) à linguagem de consulta de O2 [BaCD89].

3.3.4. Conclusão.

Como no caso do PDM, tentaremos avaliar o modelo de dados do Iris, utilizando como base as características obrigatórias definidas em [ABDD89] para os bancos de dados orientados a objetos. Por ser baseado em DAPLEX, o modelo de dados do Iris apresenta características similares ao PDM, por exemplo, os conceitos de identidade de objetos e encapsulamento são suportados de forma similar. No que respeita às outras características existem algumas diferenças.

TIPOS E CLASSES.

No Iris, todos os tipos pertencem a um mesmo tipo de objeto pai, chamado OBJECT. Isto facilita um tratamento mais uniforme de todos os objetos. Desta maneira é possível também a definição do meta-dados em termos do modelo de dados do Iris.

OBJETOS COMPLEXOS

Existem os mesmos problemas que no PDM . No Iris existem os construtores: lista, "bag" e tupla. Um dos problemas e a falta de ortogonalidade no uso destes. Por exemplo, tuplas não podem conter "bags", tuplas não podem conter tuplas como elementos e "bags" não podem conter "bags" como elementos.

COMPLETUDE COMPUTACIONAL

O Iris consegue a completude computacional, permitindo a existência de funções estrangeiras. Qualquer linguagem pode ser usada para escrever uma função estrangeira desde que o código objeto possa ser encadeado ("linked") ao processador de consultas. O problema, neste caso, é a vulnerabilidade a falhas no corpo destas funções. O Iris não tem nenhum controle sobre elas. O Iris também consegue a completude computacional, embutindo a OSQL em várias linguagens hospedeiras e através da CLI ("C Language Interface"). Nos artigos estudados não aparece nenhuma descrição destas interfaces, nem de exemplos de sua utilização, o que dificulta a sua avaliação.

POLIMORFISMO

O Iris permite os mesmos tipos de Polimorfismo que o PDM. Entretanto, o fato do Iris ter um tratamento uniforme de todos os objetos, permite simular o Polimorfismo paramétrico. É possível definir funções sobre o tipo OBJECT e por inclusão elas podem ser utilizadas por qualquer dos tipos utilizados no Iris.

3.4. Alguns Comentários sobre Modelos de Dados Funcionais para Aplicações Não Convencionais.

Nesta seção faremos alguns comentários sobre os modelos "orientados a objetos" baseados em DAPLEX. Estes sistemas têm as seguintes características:

- Os objetos não têm um estado explícito; seus atributos e relacionamentos são funções que aplicadas sobre o objeto retornam os valores associados. Sendo assim, um objeto complexo deve ser identificado por um subconjunto de todas as funções que são definidas sobre um tipo particular de objeto. A questão principal, é então : como identificar as funções que formam o objeto complexo ?, e Como diferenciar a estrutura das operações ou métodos ?

- Os objetos em Iris e no PDM são passivos talvez por razões de simplicidade e facilidade de implementação. Uma pesquisa posterior do grupo que trabalhou em PDM (PROBE) esta tentando estender este modelo para tratar objetos ativos [DaBM88]. Trabalhando com objetos passivos e necessário determinar a maneira de associar (encapsular) os objetos às operações que podem ser aplicadas a eles. Estes modelos adotam um enfoque tipo Modula-2 [Wirt83]. As propriedades de um objeto podem ser acessadas ou manipuladas passando o objeto como um parâmetro a uma operação. Porém estas operações não pertencem aos objetos ou seus tipos. O projetista do banco de dados pode esconder os detalhes de implementação das operações criando módulos [DeKL85].

- Estes modelos não diferenciam, no nível de programas de aplicação, entre essas propriedades dos objetos cujos valores são armazenados em registros no disco e essas cujos valores precissam ser calculados. Portanto, uma propriedade pode mudar de um tipo para outro, durante o tempo de vida de um banco de dados. Isto implica que existe uma maior flexibilidade e independência de dados.

- Uma fraqueza de um modelo de dados baseado em funcões e orientado a objetos, é que os relacionamentos não são bem modelados. Por exemplo, o relacionamento entre empregados e seus departamentos é modelado através da função, DEPARTAMENTO_DE(EMPREGADO). Para se obter simetria nas consultas é necessário introduzir a funcão inversa EMPREGADO_EM(DEPARTAMENTO). Estas duas funcões devem ser acopladas , já que as duas modelam o mesmo relacionamento. O problema é piorado se o usuário quiser modelar um relacionamento ternário ou n-ário, tal como no exemplo classico de peças, projetos, fornecedores [DateB1]. Neste caso o relacionamento ternario entre as entidades, cujo atributo e' quantidade, gera 16 funções interrelacionadas possíveis, e a nocão de funcão inversa não soluciona o problema. Estes relacionamentos no PDM e no Iris são modelados através de funções predicados, por exemplo:

EMBARQUE(PEÇA, PROJETO, FORNECEDOR, INTEGER) -> BOOLEAN

esta função é verdadeira se os objetos passados como parâmetros pertencem ao relacionamento. Funções tais como:

QUANTIDADE_EMBARCADA(PEÇA, PROJETO, FORNECEDOR) -> INTEGER

podem ser definidas em termos da função predicado.

4. DESCRIÇÃO DA LINGUAGEM LBF+

4.1 Introdução

Nesta seção nós pretendemos mostrar uma linguagem de programação de banco de dados interativa, LBF+, baseada no modelo funcional de Shipman e que mistura facilidades de outros sistemas como o Iris, o PDM e O2 [LeRi89]. A idéia é ter uma linguagem uniforme e concisa que permita ao usuário poder criar, operar e atualizar seu banco de dados. Esta linguagem é uma extensão da linguagem LBF [Pere88], à qual foram adicionadas algumas facilidades de orientação a objetos enumeradas na seção 3.1, além de uma linguagem tipo SQL para consultar o banco de dados. Esta linguagem SQL é um subconjunto da linguagem LBF+ o que evita a "descasamento de impedância" e a aprendizagem de uma nova linguagem.

Uma descrição detalhada da sintaxe da LBF+ pode ser encontrada no Apêndice. A seguir descreveremos suas características mais importantes.

4.2. Estruturas

LBF+ assim como DAPLEX, utiliza dois tipos de primitivas estruturais para a modelagem do mundo real: os conjuntos de objetos e valores e as funções que são aplicadas a eles para relacioná-los entre si. Entretanto, ao contrário dos outros modelos funcionais, LBF+ manipula valores estruturados e eles podem fazer parte do domínio de uma função.

A característica básica de um objeto é sua identidade, que é imutável, persiste durante o tempo que o objeto existir, e é independente dos valores de qualquer um de seus atributos (que são funções).

A estrutura dos objetos e dos valores, isto é, suas propriedades e relacionamentos entre eles, são modelados através de funções. Entretanto, os objetos encapsulam esta estrutura e também seu comportamento (operações), enquanto que os valores não. Tanto os tipos quanto as classes são criados explicitamente através do uso de comandos, mas só os tipos são parte da hierarquia de herança. O tipo mais genérico será chamado de OBJECT. Cada classe, sempre está associada a um tipo que descreve a estrutura de suas instâncias.

Para explicar melhor estes conceitos utilizaremos como exemplo, uma parte do sistema definido em [AtBu87], que consiste na descrição de um inventário de uma companhia que fabrica peças. Nós descreveremos exatamente a parte onde é representada a forma de como certas peças são feitas de outras peças. O banco de dados em LBF+, seria descrito como na Fig. 4.1.

```
Const MaxInt() -> 32767;
Type Quantidade() -> 1..MaxInt,
     Cruzeiros() -> Real,
     Gramas() -> Real;

Type Peça() -> Object
Function Struct(Peça) ->
         Tuple(Nome: String(16) Unique;
                  UsadaEm: Set(Tuple(Composicao: PecaComposta;
                                     Quant: Quantidade))
                  Derived of Uso(Peca,PecaComposta,Quantidade))
Function Operations(Peca) ->
         Tuple(CustoPeso: Tuple(CustoTotal: Cruzeiros;
                                PesoTotal: Gramas)
                          Forward);

Type Fornecedor() -> Object
Function Struct(Fornecedor) -> Tuple(CGC: String(10) unique;
                                     Nome: String(20));

Type PecaBasica() -> Peca
Function Struct(PecaBasica) ->
         Tuple(Custo: Cruzeiros;
               Peso: Gramas;
               FornecidoPor: Set(Fornecedor));

Type PecaComposta() -> Peca
Function Struct(PecaComposta) ->
         Tuple(CustoMontagem: Cruzeiros;
               AcrescimoPeso: Gramas;
               Usa : Set(Tuple(Componente: Peca;
                               Quant: Quantidade))
               Derived of Uso(Peca,PecaComposta,Quantidade)
               Total);
Function Operations(PecaComposta) ->
         Tuple(CriePecaComposta Forward Public );


Function Uso(Peca,PecaComposta,Quantidade) -> Boolean;

/* Definição de Classes */

Persistent Var Pecas() -> Set(Peca);
Persistent Var Fornecedores() -> Set(Fornecedor);
Persistent Var PecasBasicas() -> Set(PecaBasica);
Persistent Var Pecascompostas() -> Set(PecaComposta);
```

Fig. 4.1. Descrição de um banco de dados de uma companhia que fabrica peças a partir de outras peças.

4.2.1. Sistema de Tipos

O Sistema de tipos d LBF+ consiste de um conjunto de tipos básicos, pré-definidos e um conjunto de construtores de tipos. Os tipos básicos são: "integer", "real", "string", "boolean", e os tipos escalar e "subrange", utilizados em Pascal. Os construtores de tipos são: conjunto, lista e tupla ("bag" é usado mas não de uma maneira ortogonal). Os tipos estruturados são construídos recursivamente através da utilização de outros tipos estruturados e tipos básicos. Os construtores podem ser usados ortogonalmente.

Utilizando a classificação encontrada em [Beeri80], mostraremos como cada um destes conceitos é representado na LBF+ a nível do esquema:

Valores Atômicos.

Na LBF+ é possível dar um novo nome à extensão de um dos tipos básicos, ou para um subconjunto deles. Por exemplo, os tipos **Quantidade** e **Cruzeiros** da Fig. 4.1.

É possivel definir constantes como em Pascal [JeWi74]. Por exemplo, a constante **Maxint** da Fig. 4.1.

Valores estruturados.

Este tipo de construção é declarada na LBF+ conforme o exemplo abaixo:

```
TYPE Matriz( ) -> LIST(Vector) EXACTLY Maxtam;
TYPE Vector( ) -> LIST(Real)   EXACTLY Maxtam;
```

Este exemplo define um tipo Matriz de reais de tamanho Maxtam x Maxtam.

Objetos Abstratos.

Eles representam abstrações da aplicação. Nós os chamamos simplesmente de objetos e têm um estado ou estrutura, um comportamento (operações) e um "oid" (object identifier). Por exemplo, Peça, Fornecedor, da Fig. 4.1. Tanto os valores estruturados quanto os objetos são representados através de funções de zero argumentos como em DAPLEX. Sobre todos estes tipos nós podemos definir funções aplicadas a eles. No caso dos objetos abstratos, estas funções são encapsuladas em dois grupos formando tanto as propriedades (estrutura) quanto seu comportamento (operações). A estrutura dos objetos, é definida através da função STRUCT, enquanto que o comportamento deles, como veremos na secão 4.3, é definido atraves da funcão OPERATIONS.

Função STRUCT

Na LBF+ uma tupla é interpretada como um conjunto de funções, assim se uma função aplicada a um elemento (objeto ou

valor) retorna uma tupla, qualquer campo desta tupla pode ser recuperado compondo funções (veja seção 4.3.2). Esta idéia é utilizada para definir a estrutura de um objeto. A função STRUCT retorna uma tupla de funções que representam os atributos e relacionamentos dos objetos. Entretanto, para facilitar a manipulação destas funções, o acesso a elas é feito evitando-se a composição com a função STRUCT. Isto é, utilizando o exemplo da figura 4.1, para acessar o nome de uma peca nós chamamos Nome(Peca) e não Nome(STRUCT(Peca)). Este mecanismo também é utilizado para o caso de OPERATIONS. As funções que formam a estrutura dos objetos não geram efeitos colaterais e podem ser básicas ou derivadas. Elas são caracterizadas por serem funções cujo único argumento é o objeto associado. Desta maneira a estrutura dos objetos é definida explicitamente através da função STRUCT. No entanto, a definição da estrutura pode ser opcional e ela pode ser construída na medida em que sejam definidas e incluídas nela, as funções que a formam. Nós veremos mais adiante como isto é possível. Desta maneira é permitida uma evolução do esquema, isto é, nós podemos definir atributos ou relacionamentos dinamicamente durante o uso do sistema.

A LBF+ segue, portanto, o caminho de O2 [LeRi89] e diferencia entre objetos e valores. Nós achamos, assim como o pessoal de O2, que os SGBDOOs "puros", isto é aqueles que consideram tudo como um objeto, não são muito práticos. O usuario tem que definir novas classes toda vez que precissar de um valor complexo, o que resulta em um crescimento indesejável das classes. Por exemplo seja a declaração:

```
Type LugarTuristico() -> Object
Function Struct(LugarTuristico) -> Tuple(Nome: String(16);
                                         Endereco: Endereco;
                                         Descricao: String(50);
                                         CustoEntrada: Real );
```

Se a LBF+ permitisse unicamente a declaracão de objetos, **Endereco** seria um tipo de objetos. Porem, **Endereco** não e uma entidade independente que mereça ser classificada como um tipo de objetos. Cada elemento de **Endereco**, não é mais que um par de cadeias de caracteres (valores) dependentes semanticamente do lugar turístico associado. Portanto, não vale a pena a sua declaração como tipo de objeto e sim como:

```
Type Endereco() -> Tuple(Cidade: String(16);
                         Rua: String(16) );
```

Resumindo, nós achamos que dependendo da aplicação deve existir certa liberdade para que o projetista possa definir valores e objetos.

Outro aspecto interessante a ser considerado é a definicão de funções derivadas. A LBF+ utiliza a clausula FORWARD, da mesma maneira que o IRIS, para separar a especificação da implementação de funções derivadas.

4.2.2. Classes

No O2 [LeRi89], os objetos encapsulam dados e comportamento enquanto que os valores não. Na LBF+, este conceito é mantido. Entretanto, toda informação sobre hierarquias, operações e estruturas, é definida a nível de tipos, e não de classes como no O2. Na LBF+ os tipos são criados explicitamente e as classes são associadas posteriormente aos tipos. Por exemplo, na fig 4.1, Peças representa a classe, e Peça o tipo da classe.

A nível de classes também é possível a definição de Subclasses ou classes derivadas, o que permite definir hierarquias também a nível de Classes. Portanto como em Galileo [AlCO85], na LBF+ são definidos dois tipos de hierarquias IS-A:

- A hierarquia de tipos, que trata o aspecto intensional e em certo modo, o extensional, já que as classes são definidas a partir dos tipos.

- E as Subclasses que tratam o aspecto somente extensional.

Um exemplo de uma definição de Subclasses seria o seguinte:

```
PERSISTENT MULHERES( ) -> F IN PESSOAS WHERE
                                SEXO(F) = "feminino"
```

Galileo [AlCO85] permite 3 formas de definir subclasses: por subconjunto, por partição e por restrição. Um subconjunto é uma classe cujos elementos estão contidos na classe "Pai". Uma partição é como um subconjunto, porém com uma restrição adicional, seus elementos não estão incluídos em nenhuma outra subclasse da mesma partição. Uma restrição contém todos os elementos da classe "Pai" que satisfazem algum predicado. O predicado não pode ser definido sobre atributos modificáveis ou derivados. Na LBF+, unicamente classes definidas por restrição podem ser declaradas, veja o exemplo acima. Através do mecanismo de restrições é possível a definição de partições, como veremos mais à frente.

Vários autores [Beer90, Abit90] sugerem que é útil ter relações além das classes. As classes são baseadas em objetos enquanto que as relações são baseadas em valores e os dois conceitos são práticos. Beeri relaciona algumas das razões para um modelo ter relações. Entre elas podemos enumerar as seguintes:

- As respostas das consultas são geralmente relações, e a linguagem deve ser fechada baixo composição, assim relações precisam ser cidadãos do primeiro tipo.

- Tendo relações, nós podemos representar relacionamentos sem precisar criar objetos abstratos para representa-los.

- O princípio de ortogonalidade implica que se nos podemos usar

conjuntos de objetos, também deve ser possível usar conjuntos de elementos de qualquer tipo.

Na LBF+, as relações podem ser implementadas como conjuntos de elementos de qualquer tipo, mas também existem as funções predicados. Este mecanismo discutido na seção 3.4, nos permite modelar, de maneira simétrica, relacionamentos de qualquer tipo. Um exemplo da utilização de funções predicados é mostrado na Fig. 4.1, onde a função **USO** representa o relacionamento entre **Peças** e **PeçasCompostas**, cujo atributo é **Quantidade**. Esta função representa a composição das peças compostas. A partir desta função **USO**, podemos definir as funções derivadas **UsadaEm** e **Usa**. A função **UsadaEm** retorna as peças compostas onde uma peça é usada e a quantidade de elementos da peça usada na composição da peça composta. A função **Usa** retorna as peças e suas quantidades, usadas na composição de uma peça composta. A definição destas funções gera uma redundância lógica e não física.

4.2.3. Definição de Objetos e Valores

Na LBF+, o usuário pode definir tanto objetos quanto valores da mesma maneira que nas linguagens de programação convencionais ou nas linguagens que manipulam objetos complexos (ex: C02 [LeRi89], Galileo [AlC085], etc). Por exemplo, nós podemos definir constantes do tipo estruturado:

```
CONST PaulBrown -> TUPLE(Nome : "Paul" ;
                         SobreNome : "Brown";
                         DataNasc  : " 04/06/1961" )
```

a declaração define uma constante chamada PaulBrown com seu valor estruturado associado.

Também é possível declarar variáveis de qualquer tipo inclusive objetos abstratos. Por exemplo, podemos declarar uma variável x do tipo **PeçaComposta**:

```
VAR x -> PeçaComposta.
```

4.2.4 Persistência

No exemplo da Fig. 4.1 as constantes e os tipos persistem após a execução de uma sessão. Para evitar que os dados desapareçam ao término de uma Sessão, e permitir que eles sejam compartilhados por diferentes procedimentos, a LBF+ utiliza a cláusula **PERSISTENT**. Desta maneira LBF+ suporta o conceito de persistência ortogonalmente, isto é, de acordo com os três princípios definidos por Atkinson & Buneman [AtBu87]:

1) Persistência deveria ser uma propriedade de valores arbitrários e não limitada a alguns tipos.

2) Todos os valores deveriam ter os mesmos direitos de

persistência.

3) Enquanto um valor persiste, seu tipo deve também persistir.

Outro aspecto importante a ser destacado, é que qualquer objeto ou valor que é parte de um objeto ou valor persistente, também é persistente.

4.2.5. Encapsulamento

Assim como no O2 [Alta90], um objeto encapsula um valor (estrutura) e um comportamento desse valor. O valor de um objeto é obtido, usando a função STRUCT. Por exemplo, utilizando o mesmo exemplo do "O2 Programmer's Manual", se consideramos:

```
TYPE Number( ) -> OBJECT
FUNCTION STRUCT(Number) -> INTEGER;

e VAR Num -> Number;
```

nós podemos utilizar **STRUCT(Num)** para obter o valor do objeto **Num** e atribuir valores a sua estrutura, ou utilizá-la em expressões inteiras. Por exemplo:

```
IF  STRUCT(Num) > 1000 THEN
    STRUCT(Num) := 3
END;
```

A função **STRUCT** desencapsula o objeto, ela separa o valor do seu objeto. Entretanto, este desencapsulamento pode ser feito nos seguintes casos:

- com operações (métodos) ligadas à classe do objeto em questão ou à de um supertipo.

- com qualquer outra operação (método) ou procedimento se a estrutura for declarada como **PUBLIC**. Por exemplo:

```
TYPE Peça( ) -> OBJECT
FUNCTION STRUCT(Peça) -> TUPLE( ..... )  PUBLIC;
```

Também é possível declarar algumas funções da estrutura como **PUBLIC**.

O encapsulamento é um conceito polémico em BDs não convencionais. Na LBF+, a declaração de uma função de mais de um argumento pode ser parte da definição de qualquer um dos tipos de seus argumentos ou pode ser uma função independente. Segundo Dayal [Daya89], separando a definição de funções de múltiplos argumentos da definição de tipos, viola o princípio de encapsulamento porque estas funções para manipular os argumentos (no caso de serem objetos) têm que conhecer a estrutura deles. Entretanto ele considera mais natural separar a definição de funções de múltiplos argumentos dos tipos porque desta maneira os

argumentos são tratados uniformemente e muitas vezes o encapsulamento dificulta a implementação das aplicações. Nós, ao contrário de Dayal, achamos que do ponto de vistya da Engenharia de Software, é mais importante controlar o encapsulamento. Portanto por "default" todas as funções que compõem a estrutura (STRUCT) são privadas e as operações (OPERATIONS) são publicas, porém quando houver necessidade de violar o encapsulamento, o usuário poderá utilizar os comandos PUBLIC e PRIVATE para mudar esta característica.

Nem todas as funções podem ser atualizadas. Só as funções básicas ou armazenadas e algumas das funções derivadas: as "funções inversas" (declaradas através da cláusula **OPPOSITE OF**) e as funções derivadas de funcões predicados. O problema de atualizar funções derivadas, em geral, é equivalente ao problema de atualização de visões e sua discussão está fora do escopo deste trabalho.

As funções predicados, são armazenadas (básicas) e são independentes de qualquer argumento (tipo), portanto não têm encapsulamento. Sendo assim, foi definido como restrição do sistema, que uma função predicado não pode ser atualizada diretamente. Elas podem ser atualizadas criando funções derivadas associadas a um dos tipos que é argumento. Um exemplo deste caso será estudado na Seção 4.3.4.

4.3. Operações

Do ponto de vista das operações, a LBF+ apresenta diferenças com relação a outras propostas baseadas em DAPLEX e definidas ateriormente [Pere88, Kulk83, WiLH90, MaDa86]. Estas diferenças são abordadas a seguir.

4.3.1. Operações de manipulação de Dados

A LBF+ utiliza expressões e comandos como construções básicas para a manipulacão de dados. As expressões são combinações de chamadas de funcões, constantes, variaveis e operadores, respeitando sempre a compatibilidade de tipos. Elas sempre aparecem dentro de comandos e representam uma coleção dos objetos ou valores ou um só objeto ou valor. Uma colecão pode ser um conjunto, uma lista ou em certos casos um conjunto com elementos repetidos ("bag"). Os valores podem ser simples como por exemplo, um inteiro, um real, um escalar, etc., ou estruturado (agregado) como uma tupla de elementos. As expressões têm duas características: um resultado e um papel. O resultado da expressão é a coleção de elementos (ou elemento) retornado após sua avaliação. O papel de uma expressão está relacionado com o tipo através do qual os elementos do resultado são interpretados.

A LBF+ segue a proposta de O2 e Galileo e a equivalência de classes é definida pelo nome enquanto que a equivalência de tipos é dada pela estrutura. Isto é, o tipo dos valores depende

unicamente da estrutura, não adianta definir dois tipos com nomes diferentes se têm estruturas iguais. Ao contrario duas classes são sempre diferentes, e a regra de compatibilidade e dada pelo nome das classes.

A LBF+ suporta um só tipo de igualdade a "deep equality"; isto constrasta com outros sistemas onde varios tipos de igualdade são suportados ("Shallow equality") [KhCo86]. Uma desvantagem de usar "deep equality", entretanto, e seu alto custo computacional. O teste de igualdade de valores aninhados estruturalmente pode ser muito custoso e depende da profundidade da estrutura. Porém, nós achamos, da mesma forma que Dayal [Daya89], que "bons esquemas de banco de dados não contém estruturas muita profundas", e o sistema definido desta maneira, pode incentivar ao usuário na definição de objetos abstratos quando seja necessário.

A LBF+ fornece dois tipos de comandos de iteração, o FOR e o WHILE. O exemplo abaixo ilustra a aplicação do comando FOR:

```
FOR EACH Pb IN PecasBasicas WHERE Custo(Pb) > 100.00
    WRITELN(TUPLE(Nome: Nome(Pb);
                  Custo: Custo(Pb); Peso: Peso(Pb)));
```

que imprime o nome, custo e peso de todas as peças básicas que custam mais de 100.00 US$.

O segundo comando, é o comando WHILE, utilizado na maiorias das linguagens de programação. Ele foi definido para facilitar a escrita dos programas por parte do usuário.

A LBF+ também fornece dois tipos de comandos condicionais o comando IF e o comando CASE. Uma definição da sintaxe destes comandos pode ser encontrado no Apêndice B.

Os comandos anteriores constituem o corpo dos procedimentos. Estes procedimentos são funções que geram efeitos colaterais, isto é, permitem atualizar o Banco de Dados ou valores temporários. Estes procedimentos podem ser associados a algum tipo de objeto ou serem independentes. No primeiro caso, formarão parte das operações dos objetos.

A seguir descreveremos como é feita a definição das operações, mas antes falaremos um pouco sobre a especificação do papel ("role") de uma expressão e sobre composição de funções. A composição de funções é um operador muito importante dentro do modelo funcional, principalmente para construir expressões.

4.3.2 Composição de Funções

A composição de funções desempenha um papel tão importante no modelo funcional quanto a operação de junção ("join") na álgebra relacional. Na matemática a composição de funções é sempre feita a partir de funções simples. Portanto, a definição

de f(g()) é direta:

$$f(g(c)) = \{ a \mid \exists b [g(c) = b \text{ e } f(b) = a] \}$$

Em LBF+ as funções podem retornar um conjunto de entidades requerendo um tratamento especial. Assim, se f for uma função simples e g uma função multivalorada a composição é definida como

$$f(g(c)) = \{ a \mid \exists b [b \in g(c) \text{ e } f(b) = a] \}$$

isto é, o resultado é o conjunto formado pelos elementos resultantes da aplicação de f sobre cada elemento de g(c).

Se f e g forem funções multivaloradas a composição é definida como

$$f(g(c)) = \{ a \mid \exists b [b \in g(c) \text{ e } a \in f(b)] \}$$

isto é, o resultado é a união dos conjuntos retornados da avaliação da função f sobre cada um dos elementos do conjunto retornado da avaliação da função g.

4.3.3. Definição de Operações

Os objetos na LBF+, assim como na maioria do sistemas orientados a objetos, são manipulados por operações. Na LBF+ estas operações são representados através de funções e/ou procedimentos. As funções como já foi visto, não têm efeitos colaterais enquanto que os procedimentos podem ter. A cláusula **FORWARD** utilizada na definição de funções derivadas é também utilizada no caso dos procedimentos.

```
Procedure CustoPeso -> x: Tuple(CustoTotal: Cruzeiros;
                                PesoTotal: Gramas);
Using
  Var ResultParcial -> Tuple(CustoTotal: Cruzeiros;
                             PesoTotal: Gramas);
  If Self isIn PecasBasicas Then
     CustoTotal(x) := Custo(Self);
     Pesototal(x) := Peso(Self);
  Else For Each Elemento In Usa(Self)
                ResultParcial := CustoPeso(Componente(Elemento));
                CustoTotal(x) := CustoTotal(x) + Quant(Elemento)
                                 * CustoTotal(ResultParcial);
                PesoTotal(x) := PesoTotal(x) + Quant(Elemento)
                                * PesoTotal(ResultParcial);
       End;
  End;
End;
```

Fig. 4.2 Definição do procedimento CustoPeso

As operações podem ser de consulta ou de atualizacão. No caso de operações de consulta, elas podem ser representadas através de funções armazenadas extensionalmente ou derivadas. Por exemplo, seja a função **FilhosComuns** que associa duas pessoas ao conjunto de filhos que eles têm em comum (exemplo emprestado do [Beer90]). Esta função pode ser declarada em LBF+ da seguinte maneira:

```
FUNCTION OPERATIONS(PESSOA) ->
                    TUPLE(FilhosComuns(Pessoa): SET(Pessoa);
                          . . . .                          );
```

e para chamar a operação é necessário a seguinte notação:

```
FilhosComuns(P)(Q);
```

```
Persistent Type Memo() -> Object
Function Struct(Memo0 -> Tuple(ParaPeca: Peca;
                               CustoTotal: Cruzeiros;
                               PesoTotal: Gramas);
Var PecasProc -> Set(Memo);

Procedure CustoPeso -> x: Tuple(CustoTotal: Cruzeiros;
                                PesoTotal: Gramas);
Using
  Var ResultParcial -> Tuple(CustoTotal: Cruzeiros;
                             PesoTotal: Gramas);
      PecaP -> Memo;

  If Self Isin PecasBasicas Then
     CustoTotal(x) := Custo(Self);
     Pesototal(x) := Peso(Self);
  Else
     PecaP := The Memo In PecasProc Where ParaPeca(Peca) = self;
     If PecaP = Nil Then
       For Each Elemento In Usa(Self)
           ResultParcial := CustoPeso(Componente(Elemento));
           CustoTotal(x) := CustoTotal(x) + Quant(Elemento)
                            * CustoTotal(ResultParcial);
           PesoTotal(x) := PesoTotal(x) + Quant(Elemento)
                           * PesoTotal(ResultParcial);
       End;
       PecaP := New(PecasProc);
       Custototal(PecaP) := CustoTotal(x);
       PesoTotal(PecaP) := PesoTotal(x);
     Else
       Custototal(x) := CustoTotal(PecaP);
       PesoTotal(x) := PesoTotal(PecaP);
     End;
  end;
End;
```

Fig. 4.3. Nova Definição do Procedimento CustoPeso

Esta chamada pode ser pensada como o envio da mensagem, FilhosComuns, cujo argumento é a pessoa P, para a pessoa Q.

Outro exemplo de operações, neste caso um procedimento, pode ser achado nas figuras, 4.1 e 4.2. O procedimento **CustoPeso** foi definido como uma operação sobre o tipo **Peça**.

A implementação da operação CustoPeso pode ser ineficiente para o caso em que muitas peças são componentes de varias pecas, que por sua vez são componentes de uma outra peça. Neste caso o cálculo de CustoTotal e PesoTotal terá que ser repetido muitas vezes. Uma solução pode ser definir uma variável temporária à Sessão, chamada PecasProc, como é feito na Fig. 4.3.

```
procedure CriePecaComposta;
Using
  Var Compnome -> String;
      NewQuantid -> Real;
      PecaComp -> Peca;
  WriteIn("Digite os dados da PecaComposta:");
  ReadIn(CustoMontagem(self),AcrescimoPeso(self));
  WriteIn("Digite os dados do componente");
  ReadIn(Tuple(Nome: CompNome; Quantidade: NewQuantid));
  While CompNome <> " " do
    PecaComp := The Pb In Pecas Where Nome(Pb) = CompNome;
    If PecaComp <> Nil Then
      Add Tuple(Componente: PecaComp; Quant: NewQuantid)
              To Usa(Self)
    Else
      Write("A Peca cujo nome é ", CompNome, "nao existe");
    End;
    WriteIn("Digite os dados do componente");
    ReadIn(Tuple(Nome: CompNome; Quantidade: NewQuantid));
  End;
End;
```

Fig. 4.4 Criação de uma Peça Composta

4.3.4. Criação de Objetos

A criação de objetos é feita através do Comando **NEW**. O comando **NEW** toma como entrada o nome da classe ou do tipo correspondente ao objeto a ser criado. No primeiro caso, ele é incluído automaticamente na classe, enquanto que no outro caso ele deve ser incluído na classe associada, através do comando **ADD**. Por exemplo, seja a operação **CriePeçaComposta** da Fig. 4.4. Nós podemos chamar esta operação da seguinte maneira:

```
VAR  x -> PeçaComposta;
/*  Inserção Automática na classe  */
x := NEW(PecasCompostas);
CriePeçaComposta(x); /* A estrutura do objeto é preenchida */
```

ou

```
VAR x  -> PeçaComposta;
/* Inserção Manual na classe */
x := NEW(PeçaComposta);
ADD x TO PeçasCompostas;
CriePeçaComposta(x);
```

Quando o objeto é criado são associadas a ele valores "default", dependendo do tipo associado. Os valores "default" são: o "string" vazio, o inteiro 0, o real 0.0, a lista e conjunto vazio, e a tupla de valores "default".

4.3.5. Manipulação de valores

Na LBF+ podem ser definidos valores simples, ou estruturados. Os valores simples podem pertencer a qualquer tipo primitivo (ou pré-definido). Aos tipos primitivos já existentes na LBF (inteiro, real, string, booleano), foram adicionados os tipos escalar e "subrange" de Pascal [JeWi74].

Os valores estruturados são construídos através do uso de listas, conjuntos e tuplas. Por exemplo, utilizando o banco de dados descrito na Fig. 2.2, nós podemos definir o tipo Artigo da seguinte maneira:

```
TYPE Artigo( ) -> OBJECT
FUNCTION STRUCT(Artigo) -> TUPLE(Titulo: SRING FIXED;
                                 Resumo: STRING;
                                 Periodico: Periodico;
                                 Autores: LIST(Autor);
                                 Assuntos: SET(Assunto);
                                 Usuarios: SET(Usuario)
                                   OPPOSITE OF Artigos(Usuario));
```

A partir da declaração acima, nós podemos associar um valor a um objeto criado através da claúsula NEW:

```
Var ArtigoPersist -> Artigo;
ArtigoPersist := New(Artigo);
STRUCT(ArtigoPersist) :=
   TUPLE(Titulo: "Types and Persistence in Database Programming
                  Languages";
         Resumo: x;
         Periódico: CompuServ1987;
         Autores  : LIST(THE x IN Autores WHERE
                         nome (x) = "Cardelli,L.");
         Assuntos : SET(THE y IN Assuntos WHERE
                        nome(y) = "Linguagens Persistentes",
                    THE z IN Assuntos WHERE
                        nome(z) =  "Tipos de Dados");

         Usuarios : SET(THE x IN Usuarios WHERE
```

```
                                    nome (x) = "José Pérez));
```

Neste exemplo a variável ou constante x associada a **resumo** deve ser do tipo string e o seu valor deve ser um resumo do artigo. A variável **CompuServ1987**, deve ser do tipo **Periódico** e o seu valor deve ser:

```
THE z IN Periodicos WHERE Nome(z) = "Computing Surveys" AND
                          Volume(z) = 18 AND Numero(z) = 2;
```

Assuntos, **Autores**, **Usuários** e **Periodicos** são coleções de objetos ou classes definidas da seguinte forma:

```
PERSISTENT VAR Autores( ) -> SET(Autor)
```

Nós podemos atualizar a estrutura do objeto representado pela variável **ArtigoPersist** em qualquer operação associada ao objeto, ou se a estrutura for pública, em qualquer outro procedimento ou no escopo da Sessão. Por exemplo, nós podemos anexar um outro autor ao artigo ou atualizar a lista se a informação de um dos autores estiver errada:

```
ADD THE x IN Autores WHERE Nome(x) = "Buneman, P."
      TO Autores(ArtigoPersist);

Autores(ArtigoPersist)[1] := THE x IN Autores
                             WHERE Nome(x) = "Atkinson, M.",
```

A LBF+ fornece além das facilidades anteriores, operadores para concatenar listas, testar se um elemento pertence a uma lista (ISIN), extrair uma sublista de uma lista, converter um conjunto a uma lista, etc. Para conjuntos, LBF+ fornece os operadores comuns de união, diferença, interseção, cardinalidade, além dos operadores para converter uma lista em conjunto, testar se um elemento pertence a um conjunto, etc.

4.3.6. Interface SQL

A linguagem SQL é no momento a linguagem de consulta mais usada. A maioria dos sistemas que não ofereciam uma interface SQL, estão sendo estendidos para oferecê-la. Portanto, qualquer SGBDOO que deseje ser utilizado amplamente, deve fornecer uma interface SQL. Por esta razão, LBF+ facilita a definição de expressões SQL, que têm a seguinte forma:

```
SELECT Expressão Simples
FOR EACH Lista de Expressões Multivaloradas
[WHERE Predicado]
```

Este tipo de expressão pode ser usada em todos os contextos onde uma expressão multivalorada é usada (Veja Apêndice B). Por exemplo, nós podemos recuperar todos os artigos publicados por um autor e atribuir esse resultado a uma variável declarada anteriormente.

```
VAR x -> SET(Artigo);

x := SELECT Art
     FOR EACH Art IN Artigos, Aut IN Autores
     WHERE Nome(Aut) = "Atkinson, M." AND
           Art ISIN Artigos(Aut);
```

Esta forma de expressar consultas é semelhante a OSQL do IRIS [FACC89] e à linguagem de consulta do O2 [BaCO89].

4.3.7. Comentários sobre a atualização do Banco de Dados.

Na LBF+ é possível incluir um objeto existente no banco de dados dentro da classe de um determinado tipo. Para isto, é utilizada a mesma construção (comando) que é usada para a inclusão de objetos na coleção correspondente ao contra-domínio de uma função multivalorada. Por exemplo,

```
ADD THE x IN Autores WHERE Nome(x) = "Cardelli, L." To Usuários.
```

inclui o autor de nome "Cardelli, L." no conjunto do Usuários da biblioteca Pessoal. Isto é possível no caso de não existir uma restrição que indique que as coleções **Usuários** e **Autores** são disjuntas.

Uma inclusão de um objeto em uma coleção pode ser feito somente no caso do tipo do objeto e o tipo da coleção pertencerem à mesma árvore de tipos. Existe um efeito colateral nesta inclusão, que consiste na inclusão do objeto em algumas das classes associadas a tipos que são supertipos do tipo da classe em questão. Por exemplo no caso acima, x tambem seria inserido nas superclasses de **Usuários** que não sejam superclasses de **Autores**.

Da mesma maneira que nós podemos incluir um objeto em uma classe, nós podemos excluir um objeto de uma classe, isto resulta também na exclusão de todas as referências ao objeto por parte das classes correspondentes ao seus subtipos. Por exemplo,

```
REMOVE x IN Autores WHERE Nome(x) = "Cardelli, L." From Usuários;
```

exclui o autor de nome "Cardelli, L.", da classe Usuarios, entretanto ele permanece como autor. Se ele pertencer também a alguma subclasse da classe usuários (classe associada a um tipo do tipo Usuário), ele também será removido desta subclasse.

Objetos podem ser removidos da base de dados através do comando REMOVE. Por exemplo,

```
REMOVE THE x IN Autores WHERE Nome(x) = "Cardelli, L.";
```

resulta na completa eliminação do autor de nome "Cardelli, L.", do banco de dados. A remoção deste objeto implica na eliminação

de todas suas referências nos conjuntos correspondentes a todos os seus subtipos e supertipos.

4.3.8. Comandos de Entrada e Saída de Dados

Os comandos de Entrada e Saída de Dados da LBF+, são equivalentes aos da Linguagem Pascal [JeWi74]. Nós sabemos que um sistema de Entrada e Saída mais elaborado é necessário para satisfazer as necessidades dos usuários, porém, nosso proposito não era elaborar geradores de relatórios sofisticados e sim mostrar que o modelo funcional de dados pode ser estendido de uma maneira adequada para tratar aplicações não convencionais.

Os comandos WRITE e READ do Pascal são estendidos para facilitar a Entrada e Saída de expressões multivaloradas (coleções de valores). Na LBF+, nós podemos imprimir ou ler uma variável x que representa uma coleção de valores. Objetos não podem ser escritos ou lidos. Existe um formato padrão de entrada e saída que é mostrado no vídeo cada vez que o usuário deseja ler ou escrever um valor estruturado. Por exemplo, se o usuario quizer ler a estrutura de uma pessoa, vai aparecer no vídeo o "layout" da estrutura, para o usuário preencher desde o terminal. O processo inverso é feito para o comando de escrita. Um "layout" seria o seguinte:

```
TUPLE(Nome:_
      Endereco:
      Idade:
     )
```

4.4 Especificando o Papel ("role") de uma Expressão

O papel ("role") de uma expressão pode ser explicitamente especificado por meio do operador AS, como no exemplo abaixo correspondente à consulta, "achar os autores que são usuários e que têm em seu poder artigos publicados por eles".

```
FOR EACH x IN Autores WHERE
    x ISIN Usuarios AND
    EXISTS y IN Artigos(x) SUCH THAT y ISIN Artigos(x AS Usuário)
  WRITELN(Nome(x));
END;
```

Esta cláusula é útil para resolver conflitos no caso de herança múltipla. Neste exemplo Artigos(x AS Usuário) representa os artigos em poder de algum usuário x e Artigos(x) representa os artigos publicados pelo autor x.

4.5. Evolução do Esquema

A proposta Shipman [Ship81] permitia unicamente a adição de novas funções. A linguagem LBF [Pere88] aproveitou as ideias de Kulkarni [Kulk83] e ofereceu um novo operador, DELETE, para a

eliminação de funções que o usuário não deseja manter no banco de dados. A LBF+ permite estas duas facilidades mas muda um pouco sua sintaxe devido a seu enfoque mais orientado a objetos. Por exemplo, nós podemos incluir novas funcões dentro da estrutura de um objeto ou dentro das operações desse objeto. Por exemplo,

```
INCLUDE PalavrasChaves IN STRUCT(Artigo) WHERE
FUNCTION  PalavrasChaves(Artigo) -> SET(STRING)
                                    MAXIMUM 5 PUBLIC;
```

inclui um novo atributo, **PalavrasChaves**, no tipo Artigo e

```
EXCLUDE PalavrasChaves FROM STRUCT(Artigo)
```

exclui o atributo **PalavrasChaves**. Uma funcão pode ser excluida se ela não for utilizada dentro do qualquer outra funcão ou procedimento. Mudanças no esquema podem ser temporárias ou persistentes. No caso de serem persistentes, a adição de novos tipos, variáveis, constantes e funcões podem ser feitas atraves da utilização da claúsula **PERSISTENT**.

Para remover qualquer uma destas construções nós podemos utilizar a claúsula DELETE. Por exemplo,

```
DELETE Artigo( )
```

remove o tipo Artigo do esquema. Entretanto, não podem ser possíveis estas remoções se existir alguma referência a este tipo feita por qualquer outro tipo ou procedimento independente. Isto evita inconsistências.

4.6. Restrições

A proposta de Shipmam para a especificação de restricões em DAPLEX, inclui o uso das construções CONSTRAINT e TRIGGER. Embora a experiência prática ter mostrado que estes mecanismos são ineficientes no caso do banco de dados muito grandes [Brodi84], nós os utilizamos dentro da LBF+. Um estudo posterior pode pretender a análise de novos mecanismos: como manipuladores de exceções, utilizados em linguagens de programação e em Inteligência Artificial, ou pré-condições e pos-condicões associadas a cada operação. Este ponto será analisado com maiores detalhes nas conclusões.

Através da LBF+ nós podemos definir os seguintes tipos de restrições:

Restrições sobre a identificação (externo) de objetos.

Num banco de dados os usuários devem estar interessados em distinguir objetos individuais de modo que possam se referir a eles de forma não ambígua. Por exemplo, veja a Fig. 4.1, onde o atributo, Nome do tipo peça é definido como UNIQUE, o que indica que duas peças não podem ter o mesmo nome.

Restrições de totalidade.

São utilizadas quando um objeto estiver sempre associado a outro objeto no banco de dados. Por exemplo, veja a Fig. 4.1, onde a função Usa, associada a Peça Composta é definida como total, o que indica que toda Peça Composta deve ter alguma Peça que a compõe.

Restrições sobre os resultados funções.

Certas funções podem ser restritas de modo a não ter seus resultados alterados. Por exemplo, veja a declaração do tipo artigo, onde a função título é definida como fixa, o que indica que após a inclusão do título de um artigo este não pode ser alterado. A combinação das clausulas FIXED e TOTAL permite definir outro tipo de restrições, a chamada dependência de existência.

Restrições sobre a Cardinalidade.

As cláusulas EXACTLY, MAXIMUM e MINIMUM, são utilizadas para restringir o número de elementos de uma coleção (ex: classe) ou de uma função que retorna vários elementos. Por exemplo, veja a declaração da função PalavrasChaves definida acima, ela indica que o número máximo de palavras chaves de um artigo não pode ser maior que 5.

Restrições sobre Classes (Coleções de objetos).

Em geral as extensões de diferentes tipos podem ter elementos em comum, mas podem existir casos em que estes elementos não são permitidos. Por exemplo, no caso da biblioteca pessoal, quando nenhuma restrição define o contrário uma pessoa pode ser um usuário e um autor ao mesmo tempo, mas se isto não for permitido a restrição seguinte deve ser definida:

```
CONSTRAINT AutUsuario(Autores, Usuarios) -> DISJOINT
```

No caso acima, podem existir, objetos que são Pessoas mas não são Autores nem Usuários. Se nós não queremos que isso seja possível, podemos definir a seguinte restrição:

```
CONSTRAINT AutUsuario1(Autores, Usuarios) ->
                                        PARTITION OF Pessoas
```

Outras Restrições sobre resultados de Funções.

Existem outros tipos de restrições que para serem definidos são necessários mecanismos mais complexos. Por exemplo, consideremos as seguintes restrições:

- "O chefe de um departamento deve pertencer ao mesmo departamento". Esta restrição seria definida em LBF+, da seguinte maneira:

```
CONSTRAINT DepartamentoChefe(Departamentos) ->
                              FORALL x IN Departamentos SUCH THAT
                              Dept(Chefe(x)) = x;
```

- "O salário de um empregado deve ser maior que um certo salário mínimo e menor que um certo salário máximo que depende do "status" do usuário". Esta restrição seria definida em LBF+, da seguinte maneira:

```
CONSTRAINT SalarioStatus(Empregados) ->
                                    FORALL x IN Empregados SUCH THAT
                                    Salario(x) > MinSalario AND
                                    Salario(x) < MaxSalario(x);
```

MaxSalario é uma função derivada, definida sobre o tipo Empregado da seguinte maneira:

```
FUNCTION MaxSalario(x:Empregado) -> CASE Status(x) OO
                                      Chefe -> 1000000.,
                                      Analista -> 7000000.,
                                      ....
                                      END;
```

- "Um curso não deve ter mais de 45 alunos". Neste caso o sistema deve enviar uma mensagem ao usuário, da seguinte maneira:

```
TRIGGER Cheio(Cursos) -> FORALL x IN Cursos SUCH THAT
                                 COUNT(Estudantes(x)) = 45;
                         WRITE("O curso", Nome(x), "Esta cheio");
                         END;
```

4.7. Meta-Dados.

Num banco de dados, um dicionário que cataloga todos os nomes e seus significados em um dado contexto, é uma necessidade . Um dicionário de dados é o primeiro passo na especificação da informação semântica sobre a natureza do banco de dados e suas aplicações. Esta informação semântica, evita operações sem significado no banco de dados. O uso de meta-dados em bancos de dados permite a implementação de sistemas de bancos de dados auto-descritivos [MaRo86], isto é, sistemas nos quais existe um dicionário de dados integrado que é a única fonte de meta-dados para os usuários e aplicativos. Como já foi visto, os modelos funcionais facilitam o desenvolvimento de meta-dados. No caso da LBF+ é possível definir o esquema, através das primitivas da linguagem. Por exemplo, seja a definição do meta-dados descrita no apêndice A (uma representação gráfica pode ser vista na Fig. 4.5)

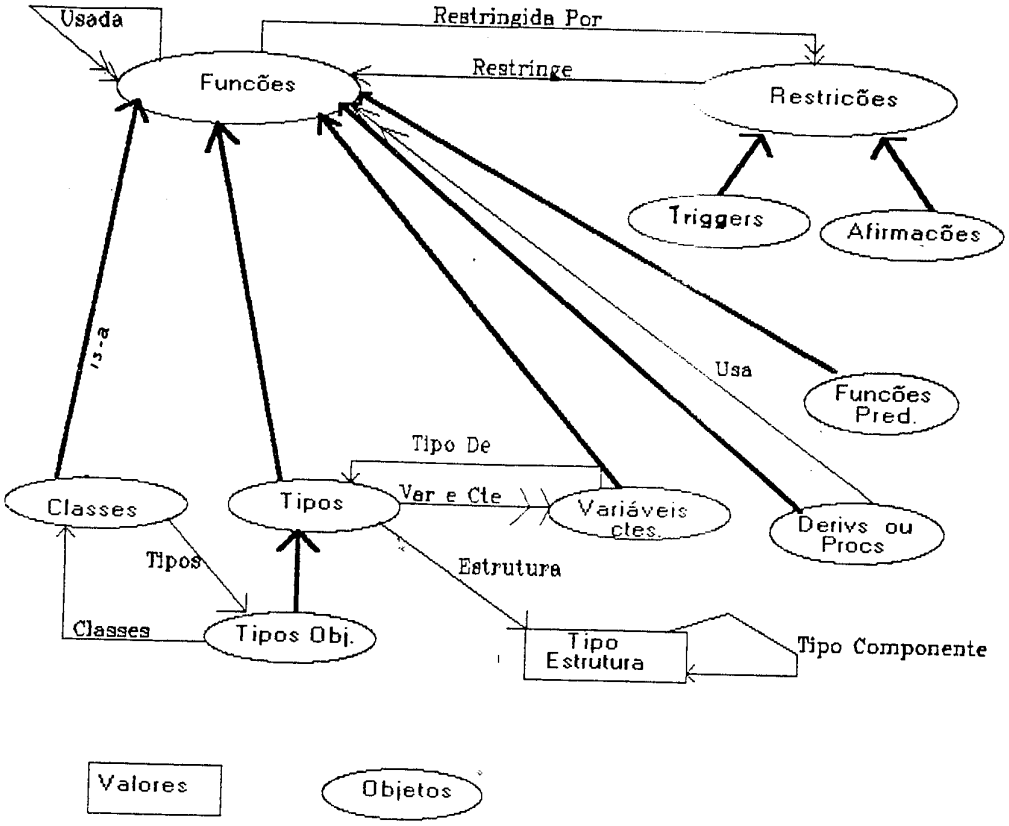

Fig 4.5 Meta-dados do Sistema

5. CONCLUSOES E PERSPECTIVAS PARA TRABALHOS FUTUROS.

Este trabalho não pretende ser a definição completa de uma Linguagem de Programação de Banco de Dados. Entretanto, nós achamos que esta proposta pode ser estendida para ser o principal linguagem (LPBD) de um SGBD de terceira geração [SRLG90].

5.1. Algumas Vantagens da Proposta LBF+

Neste trabalho mostramos que podemos estender o modelo funcional, com as características de orientação a objetos, sem perder simplicidade e uniformidade. Algumas das vantagens desta extensão são:

1. Para facilitar a utilização da linguagem, evitamos a proliferação de termos e conceitos.

2. Ao contrário de outros modelos funcionais orientados para aplicações não convencionais como por exemplo IRIS [WiLH90] e PDM [MaDa86], os objetos na LBF+ têm um estado explícito. Isto é, existe uma diferença visível entre a estrutura e as operações de um objeto. Esta vantagem, além de permitir a utilização de construtores de uma forma ortogonal facilita a utilização da LBF+ em aplicações não convencionais.

3. Através de funções predicados, a LBF+ soluciona um grande problema existente nos modelos de dados baseados em funções e orientados a objetos, que é o relacionamento entre os objetos.

5.2. Problemas em Aberto

Apesar destas vantagens alguns mecanismos importantes não foram tratados aqui para simplificar a proposta da linguagem. Entre esses temos:

- Nós não tratamos o problema de visões apesar de ser de grande importância. Entretanto, o conceito de função derivada pode ser de grande ajuda para a definição deste mecanismo. Kulkarni [Kulk83] mostra uma proposta de estender o modelo funcional para tratar visões. Um trabalho interessante no contexto da LBF+, seria incluir alguns dos mecanismos existentes em EFDM [Kulk83] para definir visões e estender posteriormente a linguagem para permitir visões atualizáveis.

- IRIS e COL [Abit90] utilizam, além das funções derivadas e armazenadas, funções externas. Estas funções externas são utilizadas principalmente para se ter completude computacional. No caso da LBF+, não seria esta a necessidade para ter implementadas funções externas. Nós achamos que as funções externas são importantes porque permitem ao usuário da LBF+, a utilização de uma variedade de subsistemas de aplicação escritos externamente (ex: Lotus 1-2-3), uma maior eficiência

(métodos gerados por compiladores mais eficientes) e a comunicação do sistema com diferentes linguagens de programação (SGBD "multilinguagem" [SRLG90]). Uma outra proposta de trabalho seria estender a linguagem para trabalhar com funções externas.

- Um mecanismo importante, que pode facilitar a modelagem conceitual, são as tuplas com campos variantes. Desta maneira a modelagem do meta-dados seria simplificada. Por exemplo,

```
TYPE TipoEstrutura(Classe: ClasseEstrutura) ->
     TUPLE(CASE Classe OF
           Conjunto, Lista -> TipoComponente : TipoEstrutura(ANY);
           Tupla ->           ComponenteTupla:
                               TUPLE(Aridade: INTEGER;
                                     EstrutTupla: SET(Funcao));
           Tipo ->            ComponenteTipo : Tipo;
           TipoPrim ->        ComponentePrim : TipoPrim );
```

Este tipo de mecanismo utilizado em Ada [Barn83] e Euclid [Lamp77] é chamado de tipo parametrizado e pode ser usado para implementar as tuplas variantes. Esta solução evita possíveis inconsistências geradas pelos registros variantes de Pascal. A cláusula ANY indica que qualquer valor pode inicializar o discriminante ("tag"). Esta solução, entretanto, deve ser estudada cuidadosamente.

- O Polimorfismo utilizado na LBF+ é bastante restrito, limita-se aos casos de inclusão e sobrecarga ("overloading"). Um estudo de um mecanismo que permita polimorfismo parametrizado, deve ser feito. A literatura ao respeito é bastante rica [Card89].

- Os tipos em LBF+ podem ser definidos de maneira que os subtipos possam refinar uma condição definida por um tipo mais geral. Por exemplo:

```
TYPE Pessoa() -> OBJECT
FUNCTION STRUCT(Pessoa) -> TUPLE(Nome: STRING(30);
                                 Idade: 1..120;
                                 Residencia: Endereco);

TYPE Empregado() -> Pessoa
FUNCTION STRUCT(Pessoa) -> TUPLE(Idade: 16..65;
                                 Supervisor: Empregado;
                                 Escritório: Endereco);
```

Neste caso a idade dos empregados é definida em uma faixa mais restrita que a idade das pessoas. Mas podem existir casos em que as hierarquias de tipos introduzam contradições [Borg88]. Por exemplo, a faixa de um atributo para um tipo especializado não é necessariamente uma faixa mais restrita daquela do supertipo, ou um atributo não é aplicável a todas as instâncias de algum subtipo. Estas exceções são tratadas por Borgida em

[Borg88]. Um trabalho interessante seria estender LBF+ para tratar estas exceções ou contradições.

- As linguagens de programação de Banco de Dados são uma boa proposta para tratar dados persistentes e não persistentes dentro de um só modelo de dados. Entretanto, existem mecanismos dentro das área de Banco de Dados e Linguagens de programação que não interagem muito bem, talvez como resultado de diferenças filosoficas existentes nas duas áreas [BIZd87]. Alguns destes mecanismos são:

Afirmações e Triggers

Estas duas ferramentas são importantes no projeto de Banco de Dados. O mecanismo de "Trigger" facilita o trabalho dos programadores de aplicação porque desenvolve funções que de outro modo o programador teria que assumir. A noção de associar restrições com os dados e declará-las só uma vez facilita também o trabalho dos programadores de aplicação. Entretanto, os "triggers" podem gerar problemas se os programadores não têm em conta quais relacionamentos são mantidos automaticamente e quais não. O programador não pode assumir que um "trigger" existe quando ele não existe ou desenvolver uma ação que de novo será desenvolvida pelo "trigger". No caso de restrições (afirmações), se elas forem violadas a transação responsável deve ser abortada, porém em alguns casos, isto é muito drástico. Portanto, é importante que os mecanismos de restrição atuem em conjunto com os manipuladores de excesões, mas a maneira como isto deve ser feito não é muito clara [BIZd87].

Outro problema com estes mecanismos é o seu funcionamento em ambientes orientados a objetos. É essencial, para que estes mecanismos sejam úteis, que uma análise estática possa ser utilizada para determinar quando necessita ser checado o predicado de um "trigger" ou de uma afirmação. Entretanto, estes sistemas são muito dinâmicos já que o usuário pode definir novas operações sobre os objetos quando ele precisar, e o sistema não pode ver automaticamente quais deveriam ser os efeitos colaterais de uma operação.

Otimização.

Em um sistema orientado a objetos não há um conjunto padrão de operações sobre os dados, já que elas são definidas para cada tipo. Por exemplo, a definição do "=" pode ser diferente para diferentes tipos. Além disso, geralmente não há forma de determinar se duas operações são comutáveis. Portanto, as técnicas padrões de otimização de consultas precisam ser revisadas. Finalmente, não existe na literatura muitos estudos sobre a utilização das técnicas de otimização de compiladores padrões na área de Linguagens de Programação de Banco de Dados.

5.3. Estado Atual do Projeto

A LBF+ está sendo desenvolvida como uma ferramenta do projeto EITIS [Melo87], em estações SUN. Como sistema de armazenamento, está sendo utilizada o sistema GEODE [Puch89] que já suporta os construtores de tipos definidos na linguagem. Está prevista a utilização do OPUS [Lanz90] para a otimização das expressões SQL.

APENDICE A
DEFINIÇÃO DO META-DADOS

```
/* Define um tipo escalar similar ao do Pascal          */
/* Otros é usado quando as funções representam          */
/* atributos relacionamentos e operações                */

TYPE TipoFunc() -> (Tipo, Classe, Variavel, Constante,
                    Procedimento, FuncaoPred, Outros);

/* Se uma função pertencer ao meta-dados seu status     */
/* sera do sistema                                      */

TYPE StatusFunc -> (Sistema, Armazenada, Derivada);

/* Define os tipos de visibilidade existentes */

TYPE TipoVisib() -> (Publica, PublicaRecupera, Privada);

/* Funcoes sao objetos com os seguintes atributos:      */
/* Nome da funcao,                                      */
/* Argumentos da funcao (uma lista de tipos)            */
/* Resultado da funcao (valor retornado pela funcao),   */
/* Tipo e status,                                       */
/* Numero de argumentos da funcao (derivada)            */
/* Conjunto de restricoes associado à funcao            */
/* Tipo de visibilidade associado                       */
/* Conjunto de funções que usam a função dentro         */
/* de sua definição                                     */
/* Documentacão da funcao (explicação)                  */

TYPE Funcao() -> OBJECT
FUNCTION STRUCT(Funcao) -> TUPLE(Nome: STRING(30);
                                 Argumentos: LIST(Tipo);
                                 Resultado : Tipo;
                                 Tipo      : TipoFunc;
                                 Status    : StatusFunc;
                                 Nargs     : INTEGER FORWARD;
                                 RestringidoPor : SET(Restricao)
                                OPPOSITE OF Restringe(Restricao);
                                 Visibilidade : TipoVisib;
                                 Usada        : SET(Funcao);
                                 Documentacao : STRING );

/*  Definição de Nargs, não precisa ser declarada    */
/*  persistente porque ela é parte de Funcao, que é  */
/*  persistente                                      */

FUNCTION Nargs(f:Function) -> COUNT(Argumentos(f));

/* Um Tipo é uma função e pode ser de objetos ou     */
/* valores. Para os dois caso sempre exixte uma      */
/* estrutura associada                               */
```

```
TYPE Tipo() -> Funcao
FUNCTION STRUCT(Tipo) -> TUPLE(Classe : (Valor, Objeto);
                               VarECte: SET(VarOuCte)
                               OPPOSITE OF TipoDe(VarOuCte);
                               Estrutura: TipoEstrutura);

/* TipoEstrutura define os construtores ou elementos    */
/* que formam o valor do tipo. Definição ineficiente    */
/* por limitações da linguagem                          */
/* Uma solucao melhor pode ser achada nas conclusões    */

TYPE TipoEstrutura() -> TUPLE(Classe : ClasseEstrutura;
                              TipoComponente : TipoEstrutura;
                              ComponenteTupla:
                                 TUPLE(Aridade: INTEGER;
                                       EstrutTupla: SET(Funcao));
                              ComponenteTipo : Tipo;
                              ComponentePrim : TipoPrim);

TYPE ClasseEstrutura() -> (Conjunto, Lista, Tupla, Tipo,
                           TipoPrim);

TYPE TipoPrim() -> (Real, Inteiro, String, Booleano, Escalar);

/* O TipoObj é um subtipo do tipo "Tipo" e além         */
/* de herdar os campos do Tipo, ele tem os seguintes    */
/* atributos:                                           */
/* Subtipos que representa todos os Subtipos imediatos; */
/* Supertipos, todos os supertipos;                     */
/* TodoSubtipos, funcao derivada que retorna todos os   */
/* subtipos do TipoObj;                                 */
/* TodoSupertipos idem ao anterior campo;               */
/* Visibilidade, define o tipo de encapsulamento da     */
/* estrutura do TipoObj;                                */
/* Operacoes, retorna o conjunto de metodos associado;  */
/* Classes, retorna as classes associadas ao tipo é     */
/* a inversa de TipoDe(Classe).                         */

TYPE TipoObj() -> Tipo
FUNCTION STRUCT(TipoObj) -> TUPLE(Subtipos : SET(Tipo);
                                  Supertipos: SET(Tipo);
                                  TodoSubtipos: SET(Tipo) FORWARD;
                                  TodoSupertipos: SET(Tipo) FORWARD;
                                  Visibilidade: TipoVisib;
                                  Operacoes     : SET(funcao);
                                  Classes: SET(Classe) OPPOSITE OF
                                                       TipoDe(Classe));

FUNCTION TodoSubtipos(x:TipoObj) -> TRANSITIVE OF
                                    Subtipos(x);

FUNCTION TodoSupertipos(x:TipoObj) -> TRANSITIVE OF
                                      Supertipos(x);

/* Classe representa as coleções de objetos e tem como  */
```

```
/* campos: a sua cardinalidade, os objetos que a formam */
/* e o tipo da Classe                                   */

TYPE Classe() -> Funcao
FUNCTION STRUCT(Classe) -> TUPLE(Cardinalidade : INTEGER;
                                 Objetos       : LIST(OBJECT);
                                 TipoDe        : TipoObj );


/*  VarOuCte representa as variáveis e constantes       */
/*  definidas  pelo usuário como parte do esquema       */
/*  e tem como único campo, o tipo ao qual pertencem    */

TYPE VarOuCte() -> Funcao
FUNCTION STRUCT(VarOuCte) -> TUPLE(TipoDe : Tipo);

/* DerivOuProc, representa as funções derivadas e os    */
/* procedimentos. Tem como campos:                      */
/* o Corpo do procedimento ou função definido em um     */
/* código intermediario;                                */
/* Parâmetros formais;                                  */
/* Funções que ele usa na sua definição.                */

TYPE DerivOuProc() -> Funcao
FUNCTION STRUCT(DerivOuProc) -> TUPLE(Corpo : STRING;
                                      ParamForm : SET(STRING);
                                      Usa : SET(Funcao) );

/* FuncaoPred, representa as funções predicados e seu   */
/* único campo retorna as funções derivadas associadas */

TYPE FuncaoPred() -> Funcao
FUNCTION STRUCT(FuncaoPred) -> TUPLE(FuncoesDeriv : SET(Funcao));

/* Restricao tem como campos:                           */
/* Nome; Um Texto no caso da restrição ter um           */
/* predicado asssociado; Os argumentos da restrição;    */
/* Uma explicação sobre a restrição; e As funcões que   */
/* ela restringe                                        */

TYPE Restricao() -> OBJECT
FUNCTION STRUCT(Restricao) -> TUPLE(Nome : STRING(30) UNIQUE;
                                    Texto: STRING;
                                    Argumentos : SET(Funcao);
                                    Documentacao: STRING;
                                    Restringe: Funcao);

/* Uma afirmação ou "assertion" é uma restrição que     */
/* tem os seguintes campos:                             */
/* Tipo que indica o tipo de restrição; Depende que é   */
/* utilizado quando o tipo é opposite, derived ou       */
/* partition; Quantidade que representa o inteiro       */
/* associado a uma restrição do tipo maximum, etc.      */

TYPE Afirmacao() -> Restricao
```

```
FUNCTION STRUCT(Afirmacao) -> TUPLE(Tipo: TipoAfirm;
                                    Depende: Funcao;
                                    Quantidade: INTEGER);

TYPE TipoAfirm() -> (unique, disjoint, partition, pred, total,
                     fixed, exactly, maximum, minimum, opposite,
                     derived);

TYPE Trigger() -> Restricao
FUNCTION STRUCT(Trigger) -> TUPLE(Corpo : STRING );

/* Definição das Classes ou Banco de Dados propriamente */
/* dito                                                 */

PERSISTENT VAR Funcoes()  -> SET(Funcao);
PERSISTENT VAR Classes()  -> SET(Classe);
PERSISTENT VAR Tipos()    -> SET(Tipo);
PERSISTENT VAR TiposObj() -> SET(TipoObj);
PERSISTENT VAR VariaveisCtes() -> SET(VarOuCte);
PERSISTENT VAR DerivsOuProcs() -> SET(DerivOuProc);
PERSISTENT VAR FuncoesPred() -> SET(FuncaoPred);
PERSISTENT VAR Restricoes() -> SET(Restricao);
PERSISTENT VAR Triggers() -> SET(Trigger);
PERSISTENT VAR Afirmacoes() -> SET(Afirmacao);
```

Todas estas funções são povoadas e atualizadas automaticamente quando os comandos de declaração e remoção de tipos, funções, restrições, são executados. Somente a funcáo documentação, aplicada aos tipos Restricao e Funcao, pode ser atualizada pelo usuário. O conteúdo destas funções pode ser recuperado através de comandos de consulta, desta maneira o usuário pode descobrir a forma do banco de dados.

APENDICE B

DEFINIÇÃO SINTÁTICA DO LBF+.

1. Notação para a descrição sintática

Para descrever a sintaxe da linguagem LBF+ foi utilizada a linguagem de especificação proposta por Wirth [Wirt77]. Esta linguagem ou formalismo é uma extensão da Forma Normal de Backus e é utilizado para se descrever da seguinte maneira (como uma metalinguagem). Nesta descrição, Simterm identifica um símbolo terminal e Simnterm identifica um símbolo não terminal).

```
sintaxe = {produção}

produção = Simnterm "=" expressão

expressão = termo {"|" termo}

termo = fator {fator}

fator = Simnterm | Simterm | "(" expressão ")" |

        "[" expressão "]" | "{" expressão "}"

Simterm = """" caráter {caráter} """"
```

Nesta linguagem segum Wirth a sintaxe é definida como um conjunto de regras sintáticas (produções) onde cada produção tem a forma Simter = expressão. Cada expressão tem a forma

termo1 | termo2 | ..... | termoN (N > 0)

o que indica que uma expressão pode produzir ou gerar um termo1 ou um termo2 ou um termoN. Cada termo por sua vez tem a forma

fator1 fator2 ..... fatorN (N > 0)

o que indica que cada termo pode gerar uma concatenação de fatores. Por último cada fator é um símbolo terminal (palavras escritas em maiúsculas e entre haspas), ou um símbolo não terminal, ou uma expressão entre parêntesis (parêntesis são utilizados com o único propósito de agrupar), ou uma expressão entre corchetes (corchetes são utilizados para denotar a união da expressão e a sentência vazia), ou uma expressão entre chaves (chaves são utilizadas para denotar a união da sentência vazia com todas as expressões resultantes da concatenacão da expressão consigo mesma tantas vezes quanto possível).

2. Sintaxe Da LBF+

```
Section = Command {Command } "END_SECTION"

Commamnd = Utilities ";"
```

```
          | Imperative ";"
          | ["PERSISTENT"] Declarative ";"

Utilities = "CONSTRAINT" Ident "(" FuncList ")" "->" ConstType1
          | "TRIGGER" Ident "(" FuncList ")" "->" Pred ";" Body
          | "INCLUDE" FuncEspec "IN" ("STRUCT"|"OPERATIONS")
            "("TypeId")" [ConstType2] ["PUBLIC"]
          | "EXCLUDE" FuncEspec "IN" ("STRUCT"|"OPERATIONS")
            "("TypeId")"
          | "DELETE" (FuncEspec | Ident )

FuncList = Ident {"," FuncList }
         | FuncEspec {"," FuncList}

ConstType1 = "UNIQUE" | "DISJOINT" | "PARTITION" "OF" Ident
           | Pred

Declarative = TypeDeclare
            | VarDeclare
            | ConstDeclare
            | FuncDeclare

TypeDeclare = "TYPE" ListDeclare

ListDeclare = Declare {"," Declare }

Declare = TypeId ["(""")"] "->" ("OBJECT"
                                 | ListTypeId
                                 | DeclConst)
          [StructDeclare]
          [OperationsDeclare]

ListTypeId = TypeId {"," TypeId }

StructDeclare = "FUNCTION" "STRUCT" "("TypeId")" "->" DeclConstr
                                                      ["PUBLIC"]
OperationsDeclare = "FUNCTION" "OPERATIONS" "("TypeId")" "->"
                    "TUPLE" "("ListFieldFunc ")"

DeclConstr = "LIST" "(" DeclConstr ")"
           | "SET" "(" DeclConstr ")"
           | "TUPLE" "(" ListField ")"
           | TypeId

ListField = Field { "," ListField }

Field = Ident ":" DeclConstr [ConstType2] ["FORWARD"] ["PUBLIC"]

ListFieldFunc = FieldFunc { "," ListFieldFunc }

FieldFunc = FuncEspec [":" DeclConstr] [ConstType2] ["FORWARD"]
            ["PUBLIC"]

VarDeclare = "VAR" ListDeclVar
```

```
ListDeclVar = DeclVar { "," ListDeclVar }

DeclVar = VarId ["("")"] "->" DeclConstr

ConstDeclare = "CONST" ListDeclConsta

ListDeclConsta = DeclConsta { "," ListDeclConsta }

DeclConsta = ConstId ["("")"] "->" SimplExpr

ConstType2 = "TOTAL" | "FIXED" | "UNIQUE"
           | ("EXACTLY" | "MAXIMUM" | "MINIMUM") int
           | "OPPOSITE" "OF" FuncEspec
           | "DERIVED" "OF" FuncEspec

FuncDeclare = (Function | Procedure)

Function = "FUNCTION" FuncDef "->" (DeclConstr
                                  | Expr
                                  |"TRANSITIVE" "OF" Expr )

Procedure = "PROCEDURE" FuncDef "->" DeclConstr "USING" Body

FuncEspec = Ident ["(" ListConst ")" ]

ListConst = DeclConstr {"," DeclConstr }

FuncDef = Ident [ "(" ListField ")" ]

Body    = Declarative ";" ListImperative "END"

ListImperative = Imperative { ";" ListImperative }

Imperative = "FOR" "EACH" MvExpr ListImperative "END"
           | "FOR" SimplExpr ListImperative "END"
           | Update
           | ("READ"|"READLN") "(" ListVar ")"
           | ("WRITE"|"WRITELN") "(" ExprList ")"
           | Selection
           | ProcCall
           | "PUBLIC" ["RETRIEVAL"] FuncEspec
           | "PRIVATE" FuncEspec

Update = (FuncCall | VarId | ListExpr "["integer "]") ":=" Expr
       | "ADD" Expr "TO" (FuncCall | VarId)
       | "REMOVE" Expr "FROM' (FuncCall | VarId)
       | "REMOVE" SimplExpr

Selection = "IF" Pred "THEN" ListImperative
            ["ELSE" ListImperative] "END"
          | "CASE" SimplExpr "OF" ListFieldCase
            ["ELSE" ListImperative] "END"

ListVar = VarId {"," ListVar}
```

```
ExprList = Expr {"," ExprList}

ListFieldCase = Fieldcase {"," ListFieldCase}

FieldCase = ListConst "->" listImperative

listConst = Const {"," ListConst }

Expr = MvExpr | SimpleExpr | CondExpr | "STRUCT" "("SimpleExpr")"

MvExpr = [Varid "IN"] MvExpr1 ["WHERE" Pred] ["AS" TypeId]

MvExpr1 = SetExpr | ListExpr | "(" Mvexpr ")"

SetExpr = SQLExpr | FuncCall | Ident
        | "SET" "(" [ListSimpleExpr] ")" | "FLATTEN" "(" MvExpr ")"
        | "LISTOSET" "(" ListExpr ")" | ListExpr "[" Integer "]"
        | "(" SetExpr SetOp SetExpr ")"

SQLExpr = "SELECT" SimpleEXpr
          "FOR" "EACH" MvExprList
          ["WHERE" Pred]

MvExprList = MvExpr { "," MvExprList }

SetOp = "UNION" | "INTERSECTION" | "DIFFERENCE"

ListExpr = "LIST" "(" [listSimpleExpr] ")"
         | OpUnList "(" ListExpr ")"
         | "(" ListExpr "CONCAT" listExpr ")"
         | "SUBLIST" "(" ListExpr "," Integer "," Integer ")"
         | MvFuncCall | Ident

OpUnlist = "HEAD" | "TAIL"

CondExpr = "IF" Pred "THEN" Expr ["ELSE" Expr ] "END"
         | "CASE" SimpleExpr "OF" ListFieldCase1
          ["ELSE" Expr ] "END"

ListFieldCase1 = Fieldcase1 {"," ListFieldCase1}

FieldCase1 = ListConst "->" Expr

SimpleExpr = Ident | FuncCall | SinglVar
           | Pred | "TUPLE" "(" ListFieldExpr ")"
           | Aexp | AggCall | "(" SimpleExpr ")"
           | ListExpr "[" Integer "]"
           | "STRUCT" "("SimpleExpr")"

ListFieldExpr = FieldExpr {"," FieldExpr }

FieldExpr = Varid ":" Expr

SinglVar = "THE" MvExpr | "NEW""("Ident")"
```

```
Pred = SvFuncCall | BoolTerm | Quant MvExpr ":" Pred
     | SimplExpr "ISIN" MvExpr

Quant = "EXISTS" | "FORALL"

AggCall = "COUNT""("MvExpr")"
        | "MAX""("MvExpr")"
        | "MIN""("MvExpr")"
        | "TOTAL""(" ( ListExpr | ["BAG" "OF"] SetExpr ) ")"
        | "AVG""(" ( ListExpr | ["BAG" "OF"] SetExpr ) ")"

BoolTerm = BoolFac | BoolTerm "OR" BoolFac

BoolFac = BoolV | BoolFac "AND" BoolV

BoolV = Bprim | "NOT" Bprim

Bprim = Aexp CompOp Aexp | Ident | Bool | "(" Pred ")"

Aexp = [ "+" ð "-" ] Uns

Uns = Term | Uns "+" Term | Uns "-" Term

Term = Fac | Term "*" Fac | Term "/" Fac
     | Term "MOD" Fac

Fac = SimplExpr [ "AS" TypeId ]

FuncCall = Ident "(" ExprList ")"

         | Message

Message = Ident "(" Expr ")" "(" SimplExpr ")"

CompOp = "<" | ">" | "<=" | ">=" | "<>"

Integer = SimplExpr

Const = Int | Real | Bool | String | "NIL"

Int = Digit {Digit}

Real = [Int]"."Int

Bool = "TRUE" | "FALSE"

String = "`"Char {Char}"'"

VarId, Ident, ConstId = Identifier

TypeId = Identifier | "REAL" | "INTEGER" | "BOOLEAN"
       | "STRING" ["("Int")"]
```

REFERÊNCIAS BIBLIOGRÁFICAS

[ABDD89] ATKINSON, M.P. & BANCILHON, F & De WITT, D & DITTRICH, K. & et. al. "The Object-Oriented Database System Manifesto". In. **PROCED. OF THE INTERNATIONATIONAL CONFERENCE ON DEDUCTIVE AND OBJECT. ORIENTED DATABASES**, 1,Kioto, Japan, 1989, p.40-57.

[ABCC83] ATKINSON, M.P. & BAILEY, P. & CHISHOLM, K.J. & COCKSHOTT, W.P. & MORRISON, R. "An Approach to Persistent Programming", **The Computer Journal**, London, 26(4):360-365, 1983.

[Abit90] ABITEBOUL,S."Towards a Deductive Object-Oriented Database language", North Holland, **Data & Knowledge Engineering** No 5, p. 263-287, 1990.

[AICO85] ALBANO, A. 7 CARDELLI, L 7 ORSINI, R. "Galileo: A Strongly-Typed, Interative Conceptual language". **ACM transactions on database Syst.**, New York, 10 (2): 230-260, 1985.

[Alta90] ALTAIR. **"The C02 Programmer's Manual"** Prototype Version 1.0.Jan. 1990. 34p.

[AnHa87] ANDREWS, T. & HARRIS, C. "Combinig Language and Database Advances in an Object-Oriented Development Environment", In: **ACM OOPSLA'87 Proceedings**, 2, Orlando, FL.,1987.

[AtBu87] ATKINSON, M. & BUNEMAN, P. " Types and Persistense in Database Programming Language". **ACM Computing Surveys**, New York, 19(2):105-190, 1987.

[BaCD89] BANCILHON, F. & CLUET, S. & DELOBEL, C. "A Query Language for the 02 Object-oriented Database System". In: HULL, R. & MORRISON, R. & STEMPLE, D. (eds.), **Proc. Workshop Database Programming Languages**, 2, San Mateo, CA., Morgan Kaufmann, 1989, Pags. 25-40.

[Banc88] BANCILHON, F. "Object-Oriented Database Systems" In: **Proceedings of the ACM SIGACT-SIGMOD-SIGART Conference on the Principles of Database Systems**, Austin, TX., May 1988.

[Barn82] BARNES, J.G.P. **"Programming in ADA"**. Addisson-Wersley, London, 1982. 340p.

[Beer90] BEERI, C. "A Formal Approach to Object-Oriented Databases". North Holland, **Data & Knowledge Engineering**, No 5, p. 353-382.

[BBKV87] BANCILHON, F. & BRIGGS, S. & KHOSHAFIAN, S. & VALDURIEZ, P. "FAD: A Powerful and Simple Database language " In: **PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATABASES**, 13, Brighton, England, 1987.

[BIZd87] BLOOM, T.& ZDONIK, S.B. "Issues in The Design of Object-Oriented Database Programming Languages", In: **ACM OOPSLA'87 Proceedings**, 2, Orlando, FL. 1987, p.441-451.

[BMOP89] BRETL, R. & MAIER, D. & OTIS, A. & PENNEY, J. & et al. "The Gemstone Data Management System" In: Kim, W & LOCHOUSKY, F.H. (eds.). **"Object-Oriented Concepts, databases, and Apllications"**, New York, ACM Press, 1989. Pags. 283-308.

[Borg88] BORGIDA, A. "Modeling Class Hierarchies with Contradictions", In: **"PROCEEDINGS OF ACM SIGMOD CONFERENCE ON MANAGEMENT OF DATA"**, 1988. Pags. 434-443.

[Brod83] BRODIE, M. L. "Association: A database abstraction of semantic modelling". In: CHEN, P.P. (ed.). **"Entity-Relationship Approach to Information Modelling and Analysis"**. Amsterdam, North-Holland. 1983, Pags. 577-602.

[Brod84] BRODIE, M. L. "On the development of data models". In: BRODIE, M. L., MYLOPOULOS, J., SCHMIDT, J. W. **"On conceptual modelling. Perspectives from Artificial Intelligence, Databases, and Programming Languages"**. New York, Springer Verlag, 1984, Pags. 21-47.

[BuFr79] BUNEMAN, P. & FRANKEL, R. E. "FQL - A Functional query language". In: **"PROCEEDING OF ACM SIGMOD CONFERENCE ON MANAGEMENT OF DATA"**. Boston, Mass., 1979, Pags. 52-58.

[Card85] CARDELLI, L. **"Amber"**, A.T&T. Bell Labs., Murray Hill, NJ., 1985. (Technical Report).

[CaWe85] CARDELLI, L. & WEGNER, P. "On Understanding Types, Data Abstractions, and Polymorphism", **ACM Computing Surveys**, New York, 17(4): 471-522, 1985.

[DaBM88] DAYAL,U & BUCHMANN, A & McCARTHY, D."Rules Are Objects Too: A Knowledge Model For An Active, object-Oriented Database System". In: **Advances in object _Oriented database Systems** 1988, p. 129-143.

[DaSm86] DAYAL, U. & SMITH, S.M. "PROBE: A Knowledge Oriented Database Management System" In: BRODIE, M.L. & MYLOPOULOS, J. (eds.) **"On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies"**. New York, Springer Verlag, 1986.

[Date81] DATE, C. J. **"An introduction to database systems. Third edition"**. Addison-Wesley, Reading, MA, 1981, 574p.

[Date83] DATE, C. J. **"An introduction to database systems. Vol. II"**. Addison-Wesley, Reading, MA, 1983, 383p.

[Daya89] DAYAL, U. "Queries and Views in an Object-Oriented Data Model". In: HULL, R. & MORRISON, R. & STEMPLE, D. (eds.) **PROC. 2nd INT. WORKSHOP DATABASE PROGRAMMING LANGUAGES**, San Mateo, CA. Morgan Kaufmann, 1989, pp. 80-102.

[DeKL85] DERRET, N. & KENT, W. & LINGBAEK, P. "Some Aspects of Operations in an Object-Oriented Database". In: BORAL, H. et al. **"IEEE Database Engineering"**, New York, 4():302-310, 1985.

[Deux90] DEUX, O. & et al. "The Story of O2", **IEEE Transactions on Knowledge Data Engineering**, 2(1): 91-108, 1990.

[DMBC87] DAYAL, U. & MANOLA, F. & BUCHMAN, A. & CHAKRAVARTHY, U. & et al. "Simplifyng Complex objects: The PROBE Approach to Modelling and Querying Them". In : **PROC.GERMAN DATABASE CONFERENCE**, Darmstadt, 1987.

[FACC89] FISHMAN, D.H & ANNEVELINK, J.& Chow, T. & CONNORS, J.W. & et al. " Overview of the Irn DBMS" In: Kim W & LOCHOUSKY, F. H. **"Object- Oriented Concepts Databases, and Applications"**, New York, ACM Press, 1989.

[Fox84] FOX, S. et al. **"ADAPLEX User's Manual"**. Computer Corporation of America, March. 1984. (Technical Report CCA-84-01).

[HaMc81] HAMMER, M. & McLEOD, D."Database Description with SDM: a Modelling Mechanism for Database Aplications". **ACM Transactions on Database Systems**, New York, 6(3):351-386, 1981.

[HeZd88] HEILER, S. & ZDONIK, S. "Views, Data Abstraction, and Inheritance in the FUGUE Data Model" In: DITTRICH, K. (ed.) **Lecture Notes in Computer Science 334, Advances in Object-Oriented Database Systems.** Berlin, Germany, Springer-Verlag, 1988.

[HoWY79] HOUSEL, B. C., WADDLE, V. e YAO, S. B. "The functional Dependency Model for Logical Database Design". In" **PROCEEDINGS OF INTERNATIONAL CONFERENCE ON VERY LARGE DATABASE**, 5, Rio de Janeiro, Brazil, Out. 1979, Pags. 194-208.

[JeWi74] JENSEN, K. & WIRTH, N. "PASCAL User Manual and Report", 2nd Ed., **Lectures notes in Computer Science**, Berlim, Springer-Verlag, 1974, p.18.

[Katz86] KATZ, R.H. & et al. "Version Modeling Concepts for Computer-Aided Design Databases" In: **"PROCEEDING OF ACM SIGMOD CONFERENCE ON MANAGEMENT OF DATA"**, 1986.

[KBCG89] KIM, & BALLOU, N. & CHOU, H. T. & GARZA, J.F. & et. al. "Features of the ORION Object-Oriented Database System". In: Kim, W & LOCHOUSKY, F.H. (eds.). **"Object-Oriented Concepts, databases, and Apllications"**, New York, ACM Press, 1989.

[Kent79] KENT, W. "Limitations of record-oriented information models". **ACM Transactions on Database Systems**, New York, 4(1):107-131, 1983.

[KePa76] KERSHBERG, L. & PACHECO, J. E. S. **"A functional data base model"**, Rio de Janeiro, Brasil, Pontifícia Universidade Católica , Fev. 1976. (Monograph in Computer Science 2/76).

[KeWa87] KEMPER, A. & WALRATH, M. "An Analysis of Geometric Modeling in Database Systems". **ACM Computing Surveys**, 19(1):45-91, 1987.

[KhCo86] KHOSHAFIAN, S. & COPELAND, C. "Object Identity" In: **ACM OOPSLA'86 Proceedings**, 1, Portland, Oregon. 1986.

[Khos89] KHOSHAFIAN, S. "A Persistent Complex Object Database language". North Holland, **Data & Knowledge Engineering**, No 3 p. 225-243, 1989.

[Kulk83] KULKARNI, G. K. **"Evaluation of functional data models for database design and use"**. Edinburgh, University of Edinburgh, 1983, 153p. (Tese de Ph. D.).

[Lamp77] LAMPSON, B.W. & et al. "Report on The Programming Language EUCLID", **SIGPLAN Notices**, 12(2), 1977.

[Lanz90] LANZELOTTE, R.S.G. **"OPUS An extensible OPtimizer for Up-to-date database Systems"**. Paris, INRIA, TU-127, 1990. (Tese de doutorado).

[LeRi89] LECLUSE, C. & RICHARD, P. "The O2 Database Programmming Languages". In: **PROCEEDINGS OF INTERNATIONAL CONFERENCE ON VERY LARGE DATABASE**,15, Amsterdan, 1989, p.411-422.

[MaCB90] MANINO, M.V. & CHOI, I.J. & BATORY, D.S. "The Object-Oriented Functional Data Language", **IEEE Transactions on Software Engineering**, 16(11):1258-1272, 1990.

[MaDa86] MANOLA, F. & DAYAL, U. "PDM: An Object-Oriented Data Model". In: **PROC. IEEE WORKSHOP ON OBJECT-ORIENTED DATABASE SYSTEMS**, 1, Asilomar, 1986, p.18-25.

[Mais88] MAKINOUCHI, A. & ISHIKAWA, H. **"The Model and Architecture of the Object-Oriented Database System JASMIN"**. Fujitsu, Ltd., Kawasaki, Japan, 1988. (Working Paper).

[MaRo86] MARK, L. & ROUSSOPOULOS, N. "Metadata Management". **IEEE Computer**, New York, 19(12):26-36, 1986.

[Melo87] MELO, R. N. "EITIS: Ambiente de ferramentas integradas para o Desenvolvimento de Sistemas Interativos". SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 1, Petrópolis, Rio de Janeiro, 1987.

[Melo88] MELO, R. N. **"Bancos de dados Nao Convencionais: A tecnologia do BD e suas Novas Areas de Aplicacao"**, VI Escola de Computacao, Campinas, SP, 1988.

[MyBW80] MYLOPOULOS, J., BERNSTEIN, P. A. & WONG, H. K. T. "A language facility for designing database-intensive applications". **ACM Transactions on Database Systems**, New York, 5(2):185-207, 1980.

[Orma84] ORMAN, L. "Nested Set Languages for Functional Databases". **Information Systems**, 9(3/4):241-249, 1984.

[Orma85] ORMAN, L. "Functions in Information Systems". **Data Base**, 16(3):10-13, 1985.

[Pere88] PEREZ-ALCAZAR, J.J. **"Projeto e Implementacao de uma Linguagem para Banco de Dados Funcionais"**. Belo Horizonte, DCC-ICEX-UFMG, Marco, 1989. [Tese de mestrado].

[Puch89] PUCHERAL, P. **"Extensibilité et Performance d'un Gérant d'Objets pour Applications Bases de Donnés"**, Paris, Universidade de Paris VI, 1989. (Tese de doutorado).

[RHDM86] ROSENTHAL, A. & HEILER, S. & DAYAL, U. & MANOLA, F. "Traversal Recursion: A Practical Approach to Supporting Recursive Applications", In: **"PROCEEDING OF ACM SIGMOD CONFERENCE ON MANAGEMENT OF DATA"**. 1986.p.166-176.

[RoSt87] ROWE, L. & STONEBRAKER, M. "The POSTGRES Data Model" In: **PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATABASES**, 13, Brighton, England, 1987. Pags. 83-96.

[Ship81] SHIPMAN, D. W. "The Functional Data Model and the Data Language DAPLEX". **ACM Transactions on Database Systems**, New York, 6(1):140-173, 1981.

[SiKe77] SIBLEY, E.H. & KERSCHBERG, L. "Data Architecture and Data model Considerations". In: **PROCEEDINGS OF AFIPS NATIONAL COMPUTER CONFERENCE**, Dallas, Texas, 1977. Montvale, N. J. AFIPS PRESS, 1977, V. 46, pags. 85-96.

[Smit81] SMITH, J. M. et al. "Multibase - integrating heterogeneous distributed database systems". In: **AFIPS CONFERENCE PROCEEDINGS NATIONAL COMPUTER CONFERENCE**, Montvale, N. J., AFIPS Press, 1981, V. 50, Pags. 487-499.

[Smit83] SMITH, J. M. et al. **"ADAPLEX: Rationale and Reference Manual"**. Computer Corporation of America, 1983, (technical Report).

[SmSm77] SMITH, J. M. & SMITH, D. C. P. "Database abstractions: agregation and generalization". **ACM Transactions on Database Systems**, New York, 2(2):105-133, 1977.

[SRLG90] STONEBRAKER, M. & ROWE, L. A. LINSDAY, B. & GRAY, J. & et.al."Third Generation Data Base System Manifeste". In: **Proc. IFIP Conference Object Oriented database Systems Analysis, Design and Construction**, 1990.

[Ston86] STONEBRAKER, M. "Object Management in Postgres Using Procedures" In: **PROC. IEEE WORKSHOP ON OBJECT-ORIENTED DATABASE SYSTEMS**, 1, Asilomar, 1986, p. 66-72.

[TsLo82] TSICHRITZIS, D. C. & LOCHOVSKY, F. H. **"Data Models"**. Prentice-Hall, Englewood Cliffs, N. J.. 1982, 381p.

[WILH90] WILKINSON, K. & LINGBAEK, P. & HASAN, W. "The Iris Architecture and Implementation", **IEEE Transactions on Knowledge and Data Engineering"**, 2(1):63-75, 1990.

[Wirt77] WIRTH, N. "What can we do about the unnecesary diversity of notation for sintactic definition?". **Communications of ACM**, 20(11):822-823, 1977.

[Wirt83] WIRTH, N. **"Programming in Modula-2"** Springer-Verlag, Berlin, Germany, 1983.

[WoKa80] WONG, E. & KATZ, R. H. "Logical Design and Schema Conversion for Relational and DBTG Databases". In: CHEN, P. P. (ed.) **"Entity-Relationship Approach to Systems Analysis and Design"**. Amsterdam, North-Holland, 1980, Pags. 344-383.

[ZdMa89] ZDONIK, S.B.& MAIER, D. (eds). **"Readings in object oriented Database Systems"**. Morgan Kauffmann, San Mateo, CA., 1989.