第6卷第5期 软件学报 Vol. 6, No. 5
1995年5月 JOURNAL OF SOFTWARE May 1995

基于类型系统的元数据模型*

陈 睿 蔡希尧

(华南理工大学计算机系,广州 510641) (西安电子科技大学软件工程研究所,西安 710071)

摘要 本文研究了程序设计语言的类型系统与数据模型的关系,认识到类型间关系是表示数据模型的一种方法.基于这一思想,提出了PCT类型描述语言及基于PCT的元数据模型,以描述多种数据模型.PCT将C++类型系统与一阶谓词演算相结合,可以形式化地描述特定数据模型所规定的多方面规范.

关键词 数据模型,面向对象程序设计,数据类型,一阶谓词演算.

在程序设计语言中,常常借用类型来排除一大类语义错误.类型之所以能够起到这个作用,是因为它包含了一定的语义信息.程序设计是用程序设计语言表示和解决应用问题的过程,而分类是人类的一种主要思维方式,有了类型,人们就可以在程序设计中方便地运用这种思维方式,因而,程序设计语言的类型系统被作为表达应用领域概念和概念间关系的有效工具.另一方面,数据库系统都建立在特定的数据模型之上.数据模型是表示数据及其关系的方法,一种数据模型通常与特定的数据结构相关联.人们认识到,一种数据模型规定了只能用特定的方式来表现应用领域问题,而大多数据库系统通常只支持一种数据模型,因而数据库系统所支持的应用领域是有限的.

同一个系统能够支持多种数据模型,可使这一系统适用于更多的应用领域.为达到这个目标,Date最早在1982年首次提出了"统一数据库语言"UDL(Unified Database Language),UDL是关于数据库程序设计语言的第一个设想[1].UDL企图找到关系、网状和层次模型的共同数据结构,以便使用这一结构来同时支持3种数据模型.换言之,UDL是要以一种数据模型来支持(或综合)多种数据模型.这一思路对于传统的3种数据模型也许适用,但随着新的应用领域不断出现,越来越多的新语义概念被加入到数据模型中(称为语义数据模型[2]),用一种数据模型综合多种数据模型的想法显得不切实际.1989年,Atkinson提出了利用类型系统本身的灵活性来支持多种数据模型的设想,他称之为"类型炼金术"(type alchemy)[3].文献[3]对"类型炼金术"没有进一步的实施设想,其它文献也没有涉及此问题.然而这一想法并非异想天开,我们认为,如果有合适的类型系统,"类型炼金术"可能成为现实.在本文中,笔者将讨论一个基于类型系统的元数据模型,即描述数据模型的模型.

* 本文1993-04-19收到,1993-09-13定稿

本文由国家教委博士点基金支持.作者陈睿,1968年生,讲师,主要研究领域为软件工程,数据库系统,面向对象系统.蔡希尧,1926年生,教授,博士导师,主要研究领域为信息系统工程.

本文通讯联系人:陈睿,广州 510641,华南理工大学计算机系软件室

1 类型和数据模型

类型是程序设计语言的一种重要的抽象机构.传统的数据库系统也存在一个类型系统,只不过数据库系统的类型系统更侧重于类型的数据模型化能力,而较少关心类型对于函数施用正确性验证的作用.传统数据库系统的类型系统与程序设计语言的类型系统一样,也包含一些基本数据类型和一些类型构造符.两者在基本数据类型方面的差别不大,一般都包含字符、整数、浮点数等基本类型,这些基本数据类型是不可分解的,也就是说,它们是非结构化的,它们的值是原子值.二者的区别主要表现在构造新数据类型的能力上.我们看一个结构化类型 person 的例子.我们希望,person 的每个实例对象能够包含姓名、年龄、性别和子女的信息.传统程序设计语言通常提供了类型构造符来满足这种要求,例如 Pascal 语言的 record 和 C 语言的 struct.person 可以用 C 语言的 struct 机构定义如下:

```
struct person {char name[20];
               char sex;
               int age;
               person *children[20];};
```

然而在数据库系统中并非如此.一个数据库系统通常支持,而且只支持一种数据模型.数据库模型对类型构造能力进行了一定的限制.例如关系模型,我们可以把关系模式看成是一种构造数据类型,只不过每一个关系模式的属性必须具有基本数据类型.因为这条限制,我们不能把关系模式定义成 person 那样的类型,因为 person 类型的属性 children 的类型不是基本数据类型.person 类型必须被定义为两个关系模式.下面我们用 SQL 语言给出相应的关系定义:

```
CREATE TABLE PERSONS (PNO CHAR (50) PRIMARY KEY,
                      PNAME CHAR(20), AGE INTEGER,SEX CHAR
                      (1))
CREATE TABLE CHILDREN (CHILD CHAR (50),PARENT CHAR (50))
```

其中,所有人的记录存放在关系 PERSONS 中,父母和子女的关系则存放在 CHILDREN 中.可见,关系模型对类型构造的限制使得定义新类型时不能直接反映对象间的语义联系,导致了类型构造的复杂化,最终使查询过程复杂化.为了放宽类型构造的限制,一些新的关系系统,例如 POSTGRES[4]允许关系模式的属性是构造数据类型,问题似乎得到了解决.

然而,当考虑将程序设计语言与数据库集成起来时,又产生了新的问题:程序设计语言和数据库系统各有各的类型系统,建立在两个不同类型系统上的数据库程序语言终究不可能实现"无缝集成"(seamless integration)的目标.早期的面向对象数据库系统有一些是采用两个类型系统的,例如 O2[5]、Vbase[6]等,而最近大多采用唯一的类型系统[7~12].

数据库程序设计语言采用唯一的类型系统意味着,这一类型系统既要起到类型检查的作用,又要能够直接支持特定的数据模型.因此,为了建立一个好的类型系统,就不得不进一步研究类型系统和数据模型的关系.

这里,笔者先给出如下的定义,一个数据模型 M 包括:

· 一组基本对象类型和类型构造符,通过基本对象类型和类型构造符可以构造任意复

杂的对象类型;

- 一组语义概念;
- 一组操作;
- 一组完整性约束.

数据模型的作用是多方面的,其中一个主要的作用就是指导数据库设计,进一步讲,数据模型必须为数据库模式设计提供一套规范.一个数据库库模式可以用一个或一组结构化类型来表示,这已是一个共识[3].数据模型可以通过一组类型构造约束来指导数据库库模式设计,甚至能够基于类型系统构造数据库库模式设计工具以帮助设计最小冗余的数据库库模式,可以通过类型信息来支持模式演变[13,14].至今为止,人们对数据模型与类型系统关系的认识仅限于此.因而,数据模型中的语义概念、操作及完整性约束常常独立于类型系统而定义.这使得现有的数据库程序设计语言都只支持一种特定的数据模型,甚至于两个系统采用同样的类型系统却支持不同的数据模型.如 Exodus[15]和 ODE[16]都采用 C++类型系统,却各有各的数据模型.

2　类型的数据库建模能力

类型在数据库和程序设计语言这两个不同的背景中扮演了不同的角色:在程序设计语言中用于声明和创建对象和进行操作的语义正确性检查,在数据库中作为模式的基本构成单元.在程序设计语言与数据库的集成系统中,对类型的理解非常混乱,导致了类型理论的许多争论性问题.在这一节中,笔者给出了对一些问题的见解,下一节讨论从这些观点出发所提出的元数据模型.

2.1　类型、类和外延(type, class & extension)

现在假设 person 类型有 5 个不同的实例 p1,p2…,p5,它们都是持久的,在某一时刻,不存在其它同类型的持久对象.有这么一个问题:person 的外延如何?要回答这个问题,首先要回答"类型和类有何区别".

一种广为接受的观点认为,类型反映对象的结构特性和行为特性,而类与一个类型及一个现存该类型的所有实例所构成的集合相关联[5,13,16].换言之,类型是一个"对象工厂",被用于创建实例对象,而类则既是一个"对象工厂"又是一个"对象仓库",类隐含地指称它的所有实例所构成的集合.因而,把 person 当作一类型看时,它的外延是一个可数无限集,把 person 当作一个类看时,它的外延是{p1,…,p5}.

关系模型中的关系模式是一个类还是一个类型?一个关系 R 是一个类,与其相关联的类型是 R 的关系模式.这样看来,PERSONS 是一个类,它的类型是

(PNO CHAR(50), PNAME CHAR(20), AGE INTEGER, SEX CHAR(1))

许多面向对象的数据库系统都是采用这种观点的.例如 Orion、GemStone 和 ODE 都把类与一个实例集合隐含地关联起来,用类名来指明查询的范围.

在 O2 中,类可以通过显式的语法指令与实例集合自动关联.例如

```
add class person with extension
    type tuple(name:string,
               sex:char,
```

```
age:integer,
children: set(person)).
```

于是,类 person 的类型是一个元组构造类型,外延是所有该类型的持久实例. 关键词 with extension 的作用等效于以下的命令:

```
add name person: set(person).
```

即定义了一个类型为 set(person)名为 person 的持久对象.

在数据库程序设计语言中,类既是类型又是实例集合的观点给类型检查带来了困难[17]. O2 尽管没有彻底解决这个问题,却迈进了一大步. 我们认为,类作为"对象仓库"的作用可以用其它更好的方式来完成,而不必付出使类和类型的外延相矛盾的代价. 例如对于类型 person 的 5 个实例 p1,…,p5,可以创建一个集合来容纳它们,也就是说,类的外延可以用一个对象来表示. 例如,用 O2 命令

```
add name persons: set(person).
```

可以创建一个集合 persons,将每一个生成的持久 person 类型实例加入其中,persons 就是 person 的外延.

取消类的"对象仓库"作用,使得类和类型的概念统一起来,这样,对查询进行类型检查就非常容易,更重要的是,类指称数据库的语义被取消,"数据库也是一个对象"的命题就顺理成章了.

2.2 用面向对象的观点看待数据库

在类是"对象仓库"的语义下,面向对象数据库是一个隐含的集合,这对应于类的外延. 从外表上看,类既是数据库,又是数据库模式. 这一矛盾是产生混乱和争论的根源. 取消类的"对象仓库"作用,使我们能用面向对象的观点来看待数据库:数据库也是对象. 于是,一切成功的面向对象方法都能直接应用于数据库管理.

数据库是一个对象,它也有类型,例如,persons 的类型是 set(person),因而可以对数据库实施类型检查;对数据库的所有操作,例如插入、删除、修改、查询都可以定义为方法,于是可以通过消息传递进行数据库操作;对象的封装性有助于保证数据库的完整性和安全性;继承性可以帮助数据库功能的增量开发;基于对象的并发程序设计方法可用于实现分布式数据库和并发事务,如此等等,不一而足. 所有这些好处都是以面向对象方法看待数据库而带来的.

3 基于类型的元数据模型

数据库模式是类型,因而,数据库模式的设计规范可以通过类型间关系来确定和描述. 把数据库看成对象,数据库上的操作就可以定义在特定类型中或者用类型化的函数来描述. 数据模型中的语义概念,有些本来就是建立在类型间关系的基础上的,例如聚合(aggregation)语义关系建立在元组构造类型与属性域(即属性的类型)间关系的基础上,概括(generalization)和特化(specification)语义建立在子类型关系基础上;另一些语义概念,诸如版本,依赖性与共享性等,也常常被作为类型定义的组成部分. 数据库完整性约束,也可以在类型定义中给出,例如 O++[16]就是这样作的. 综上所述,数据库模型的各个方面,都可以用

类型和类型间关系来描述,因而,可以在类型的基础上建立元数据模型.

3.1 类型描述语言 PCT

我们将实用性和形式化两方面的要求结合起来考虑,提出了一种基于 C++类和模板(template)以及一阶谓词演算的类型描述语言,以描述参数化类型、数据库语义概念和约束.这个类型描述语言称为 PCT(Parametric Constrainted Types).

在 PCT 中,用户定义的类型可以与一个谓词集、一个约束集和一个方法集相关.PCT 提供了 int、char、float 等基本数据类型,以及一个 tuple 类型构造符.由于可以定义参数化类型,用户可以定义新的类型构造符,参数化类型实质上就是类型构造符[18].

定义谓词集的语法格式是 Predicates {谓词表}

其中,谓词表由一系列谓词构成,每个谓词间以“,”隔开.谓词表中可以包含两种谓词,一种称为“语义谓词”,语义谓词表示数据模型的语义概念,它们在类之外定义,如果某个语义谓词 P 出现在类型 T 的谓词集中,表明 T 的实例满足 P.另一种谓词称作“约束谓词”,约束谓词在类型的谓词集中定义,是为了方便书写约束的谓词演算公式,它们可被看成一种用逻辑程序设计语言书写的方法.

定义约束集的语法格式是 Constraints {约束表}

其中,约束表由一系列约束构成,每个约束词以“,”隔开.每一个约束都是一个合式的谓词演算公式,如果约束 C 出现在类型 T 的约束集中,就表明 T 的每个实例必须满足 C.

方法集由 Methods {方法外观表}给出,方法外观(signature)表由一系列方法外观构成,每个方法外观由方法名、参数类型和返回值类型构成,方法外观之间以“,”隔开.

PCT 还允许显式定义子类型,并支持多重继承性.定义子类型的语法格式是

子类型名:父类型表

其中父类型表由一系列用户定义类型构成,参数化类型作为父类型时必须首先实例化,即指明类型参数.子类型继承父类型的一切特性,包括表示、方法集、约束集和谓词集.我们假设多重继承情况下不发生冲突,以简化问题.

现在我们定义一个参数化类型 set.

```
type set〈T〉 =tuple {…}
        Methods {∪:set〈T〉 → set〈T〉,        //并
                 ∩:set〈T〉 → set〈T〉,        //交
                 −:set〈T〉 → set〈T〉,        //差
                 ∈:T→Boolean,                //元素包含
                 insert:T→ set〈T〉,          //插入元素
                 delete:T→ set〈T〉,          //删除元素
                 ⊆:set〈T〉 → Boolean         //集合包含
                 }
        Predicates {in (x,S)⇌∃ y(y∈S∧identical(x,y))}
        Constraints{∀ x(in(x,self)→type(x)≤T)};
```

我们不关心 set 的物理表示.在谓词集中,定义了一个约束谓词 in(x,S),它的逻辑值依赖于方法∈的返回值和谓词 identical 的值.谓词 identical(O1,O2)判定对象等同,它的语义

是,如果 O1 与 O2 的对象标识相同,则 identical(O1,O2)为真,否则为假. 文献[19]中给出了等同谓词 identical(O1,O2),浅相等谓词 shallow_equal(O1,O2),和深相等谓词 deep_equal(O1,O2)的定义,我们在此可直接利用. set〈T〉的约束集给出了一条约束,它表明一个类型为 set〈T〉的集合的所有元素的类型是 T 或 T 的子类型. 这条约束中的 self 是一个伪变量,其语义相当于 C++的 this 和 Smalltalk 的 self,type(O)指称 O 的类型,谓词 T1≤T2 的值当 T1 是 T2 的子类型或者 T1 与 T2 是同一类型时为真,否则为假. 上述讨论是基于文献[19]所给的对象模型的. 另外,再引入一个特殊对象标识 NIL,它指称未定义对象.

我们来讨论 set 的一些性质. 首先,用 set〈T〉可以声明同质集合,其所有元素的类型都是 T;还可以声明异质集合,其某些元素可以持有 T 的子类型. 其次,T1≤T2 并不意味着 set〈T〉≤set〈T2〉,这与 O2 的观点不同[5]. 我们不采用类型推理来推导隐含的子类型关系,在 PCT 中,子类型关系是显式定义的,要得到 set〈T1〉≤set〈T2〉关系,必须显式地写出

```
type set〈T1〉:set〈T2〉;
```

现在我们定义另一个谓词 attribute(A,T),当 T 是一个元组构造类型,A 是 T 的属性时,attribute(A,T)为真,否则为假. 例如

```
type T=tuple {A1 : int,
              A2 : char; } ;
```

则 attribute(A1,T)为真,attribute(A3,T)为假. 另外定义一个谓词 Attribute(A,O),其中 A 是一个属性名,O 是一个对象,Attribute(A,O)⇆attribute(A,type(O)). 再定义类型 T 的成分类型(componenttype)T′如下:T′是 T 的成分类型,记作 T′◁T,当且仅当对 T′的任意属性 A 都有 attribut(A,T),且 T 和 T 的 A 属性域相同,即

$$T' \lhd T \rightleftharpoons \forall A(attribute(A,T') \rightarrow attribute(A,T)) \wedge \forall X \forall Y(type(X)=T' \wedge type(Y)=T \rightarrow type(X.A)=type(Y.A))$$

两个元组构造类型 T1 和 T2,若没有同名属性,可以定义它们的复合类型(Composite type)T 如下:T 是 T1 和 T2 的复合类型,记作 T=T1⊕T2,当且仅当 T 包含 T1 和 T2 的所有属性,且不包含其它属性,即

$$T=T1 \oplus T2 \rightleftharpoons \neg\exists A(attribute(A,T1) \wedge attribute(A,T2)) \wedge \forall A(attribute(A,T) \rightleftharpoons attribute(A,T1) \vee attribute(A,T2))$$

显然,若 T1≤T2,则 T2◁T1;但逆命题并不一定成立,而 T=T1⊕T2 并不隐含 T、T1 和 T2 之间有什么子类型关系. 若 T2◁T1,我们可以在 T2 中定义一个方法,它接受 T1 类型的参数,而返回一个 T2 类型对象,我们还可以给这个方法赋予一定的特殊性,象 C++的构造函数那样,使调用此方法的语法形式不同于一般的消息传递格式. 例如

```
type T1=tuple{A1:int;
              A2:char;};
type T2=tuple{A1:int;}
        Methods{T2:T1→T2};
```

于是 T2 可以起到类型转换的作用. 对于函数

```
void f(T2 X);
```

当调用 f 的实际参数为 T1 类型时,C++的类型系统可以自动将实际参数变成 T2 类型的.

若 T=T1⊕T2,我们也可以在 T 中定义接受 T1 和 T2 参数而返回 T 类型对象的构造函数.例如

```
type T1=tuple {A1:int;};
type T2 =tuple {A2:char;};
type T  =tuple {A1:int;
                A2:char;}
        Methods {T:T1×T2→T};
```

当定义了上述的构造函数之后,可以进行对象过滤和对象组合.若 T2◁T1,O 是一个 T1 类型的对象,则 T2(O)可生成一个 T2 类型的对象,它保留了 O 中的一部分信息,过滤掉其它信息.若 T=T1⊕T2,O1 是一个 T1 类型对象,O2 是一个 T2 类型对象,则 T(O1、O2)生成了一个由 O1 和 O2 组合而成的对象,它具有 T 类型.对象过滤和对象组合对进一步讨论数据库投影、连接等操作很有意义.

3.2 元数据模型

在 PCT 基础上可以建立一个元数据模型.现在,我们可以把 M 定义为:

- 一组 PCT 类型,类型构造符可定义为 PCT 的参数化类型;
- 一组语义谓词,对象的语义可以定义在 PCT 类型的谓词集中;
- 数据库操作可以定义在 PCT 类型中,也可以定义为多态函数;
- 数据库完整性定义在 PCT 类型约束集中.

4 用元数据模型描述数据模型的实例研究

现在我们来讨论如何用上述元数据模型来描述实际应用中的数据模型.

4.1 关系模型

文献[1]所示的第一范式关系模型指出,关系模式是一个元组类型,关系模式中的每个属性域都必须是基本数据类型,关系模式的某些属性构成候选关键字,一个关系模式可以有多个候选关键字,这些候选关键字中有一个是主关键字.关系是关系模式的许多实例——元组构成的集合,每一个元组的主关键字中不能出现空值,即实体完整性约束.关系模型还要求多个关系满足引用完整性约束,即一个关系 R1 的主关键字 K 是另一个关系 R2 的外关键字,R2 中若有一个元组 t1 的 K 值不为空,则 R1 中必存在一个元组 t2 使 t2 的 K 值与 t1 的 K 值相等.

现在我们为关系模型定义两个语义谓词:

(1)Prikey(T,R)

设 R 的关系模式为 T′,则 Prikey(T,R)为真时 T◁T′且 T 是 R 的主关键字;

(2)References(T,R1,R2)

设 R1 和 R2 的关系模式分别为 T1 和 T2,则 References(T,R1,R2)为真时 T◁T1,T◁T2 且 T 是 R1 的外关键字,是 R2 的主关键字.

定义一个 PCT 类型 Rel_schema:

```
type Rel_Schema=tuple{ }
                Constraints{∀ A (Attribute (A, self)→type (self. A)=int∨ type
```

(self. A)=float ∨ … ∨ type (self. A)=char)};

上面定义表明,Rel_schema是一个元组构造类型,其实例的所有属性值都持有基本数据类型. Rel_schema是一个抽象类型,因为没有定义任何属性. 我们可以把任意关系模式定义为它的子类型,以继承它的约束集.

现在,定义一个参数化类型Relation:

type Relation〈T〉:set〈T〉

Predicates{Ckey (T′,R)⇌∃ S(type(R)=Relation(S) ∧ T′◁S) ∧ ∀ X∀ Y (in(X,R) ∧ in(Y,R) ∧ ¬identical(X,Y)→¬deep_equal(T′(X),T′(Y)))}

Constraints{(1)T≤Rel_schema,
(2)Prikey(T′,self)→∀ X∀ A(in(X, self) ∧ attribute(A,T′)→¬identical(X. A,NIL)),
(3)∃ T′(Ckey(T′,self) ∧ ∀ X∀ A(in(X,self) ∧ attribute(A,T′)→¬identical(X. A,NIL))),
(4)Reference(T′,R1,R2)→∀ X(in(X,R1) ∧ ∀ A(attribute(A,T′)→¬identical(X. A,NIL))→∃ Y(in(Y,R2) ∧ deep_equal(T′(X),T′(Y)))};

在Relation的谓词中,定义了一个约束谓词Ckey(T′,R),若T′是R的一个候选关键字,则Ckey(T′,R)为真,否则为假. 约束(1)表明,任意关系模式都是Rel_schema的子类型;约束(2)表明,在显式定义了主关键字时必须满足实体完整性约束;约束(3)表明,任意关系都至少有一个候选关键字,在没有显式定义主关键字时,必须有一个候选关键字满足实体完整性约束,若显式定义了主关键字,约束(3)自然满足;约束(4)表明,多个关系应满足引用完整性约束.

上面给出了第一范式关系模型的PCT描述. 在实际应用中,可以把每个关系模式定义成Rel_schema的子类型,把每个关系定义成Relation〈T〉或其某个子类型的实例,其中T是关系模式. 我们还可以描述第二、三、四、五范式的关系模型. 随着规范化程度的不断提高,就要求越来越多的约束,第二范式关系必定是第一范式关系,而第三范式关系必定是第二范式关系,依次类推. 因而,我们可以把第二范式关系定义为第一范式关系的子类型,它除继承了第一范式关系的所有约束外,又定义了新的约束和谓词. 同理,第三范式关系可定义为第二范式关系的子类型,依次类推.

4.2 Orion模型

Orion的数据模型是一种典型的面向对象模型. 除了支持核心的面向对象模型概念,例如继承性之外,它还支持复杂对象的共享语义和数据依赖语义,以及版本概念[13].

Orion数据模型支持4种复杂对象复合引用语义(composite reference semantics):独立共享的复合引用;依赖共享的复合引用;独立互斥的复合引用;依赖互斥的复合引用.

例如有一个类型T:

type T = tuple { A1:T1;……};

如果A1是一个独立共享的复合引用属性,则可以存在两个不同的T类型实例t1和t2,

t1和t2的A1属性值共享一个T1类型对象,即identical(t1.A1, t2.A1),且这个T1类型对象的存在不依赖于t1或t2的存在,即t1和t2被删除时它仍可存在;若A1是依赖共享的,则t1.A1或t2.A1的存在依赖于t1或t2的存在,一旦t1或t2被删除,t1.A1或t2.A1也要被删除.若A1是独立互斥的,则t1和t2的A属性值不能是同一对象,即identical(t1,t1)→identical(t1.A1,t2.A1),但t1.A1或t2.A1可以独立于t1或t2存在;若A1是依赖互斥的,则t1.A1或t2.A1的存在依赖于t1或t2的存在.为表示上述语义,我们定义两个谓词:

(1)Dependent (A,T),若A是T的属性且A是依赖的复合引用时,Dependent (A,T)为真;

(2)Exclusive (A,T),若A是T的属性且A是互斥的复合引用,Exclusive(A,T)为真.

4种复合引用语义可以用这两个谓词表示:

·依赖互斥:Exclusive(A,T)∧Dependent(A,T);

·独立互斥:Exclusive(A,T)∧¬Dependent(A,T);

·依赖共享:¬Exclusive(A,T)∧Dependent(A,T);

·独立共享:¬Exclusive(A,T)∧¬Dependent (A,T);

Orion模型的对象版本语义很复杂,由于篇幅所限,本文不予讨论,详见文献[20].

Orion中类有"对象仓库"的作用,因而每一个类都与一个该类实例集合隐含相关,换言之,用户定义好类之后,系统就隐含地生成一个集合,设类名为T,不妨将此集合记作Extension(T).若T1≤T2,则Extension(T1)⊆Extension(T2),这就是Orion的继承性语义.

下面我们用元数据模型来描述Orion的数据模型.

```
type Orion_schema=tuple { }
                    Constraints {……}
```

约束集定义了Orion模型的对象版本语义.Orion_schema是一个抽象类型,Orion的类被定义为它的子类型.现在为类所指称的"对象仓库"定义一个类型:

```
type OrionBase⟨T⟩:set ⟨T⟩
    Constraints {(1)identical (self, Extension(T)),
                 (2)T≤Orion_schema,
                 (3)∀ X∀ A(attibute (A.T)∧in (X. self)∧¬identical(X.A,NIL)
                 ∧type (X,A)≤Orion_schema→∃ y(in(y,Extension (type(X.
                 A))∧identical (X,A, Y))),
                 (4)∀ A∀ Y∀ X(Dependent(A, T)∧T′=type(A) ∧T′≤Orion_
                 schema∧deleted(x,self)→deleted(x.A,Extension(T′)))
                 (5)∀ X∀ Y∀ A(in(X, self)∧in(Y, self)∧¬identical(X,Y)∧Ex-
                 clusive(A, T)→¬identical(X. A, Y.A))),
                 (6)∀ T′∀ X(T′≤T∧ in (X, Extension(T′))→in(X, self)))};
```

文献[20]中还有更多的具体实例,本文不再一一列举.

5 小 结

由于应用要求的多变性，人们希望数据库系统能够支持多种数据模型，这在以前是很难作到的. 笔者在考察了类型和数据模型的关系之后认识到，类型间关系可以充分地表示数据模型，从而提出了元数据模型的思想. 更进一步，又提出了将数据库当作对象来看待的思想. 本文将C++类型系统与一阶谓词演算相结合提出了PCT类型描述语言，PCT类型描述语言可以形式化地描述特定数据模型所规定的诸方面规范(对象类型、语义特性和完整性约束等). 实例研究表明，我们提出的元数据模型具有足够的表示能力. 如果一个数据库程序设计语言具有PCT这样的类型系统，它就能成功地支持多种数据模型，因而可适用于更多的应用领域.

另外，由于持久程序设计语言支持类型完全和类型正交的对象持久性，基于类型系统的元数据模型和数据库作为持久对象的思想有助于将持久程序设计语言自然地扩展为支持多种数据模型的数据库程序设计语言.

参考文献

1 Date C J. An introduction to database systems, Vol. Ⅰ—Ⅱ. Addison—Wesley, 1983.

2 Peckham J *et al*. Semantic data models. ACM Computing Surveys, 1988, **20(3)**: 153—190.

3 Atkinson M. Questioning persistent types. Proc. of The 2nd Int. Workshop on Database Programming Languages, 1990.

4 Stonebraker M, Rowe L A. The design of POSTGRES. Proc. ACM—SIGMOD Conf. on Management of Data, 1988.

5 Deux O *et al*. The story of O2. IEEE Trans. on Knowledge and Data Engineering, 1990, **2(1)**: 91—108.

6 Damon A. C++ and cop: a brief comparison. In: Gupta R, Horowitz ed. Object—Oriented Databases with Applications to CASE, Networks and VLSI CAD, Prentice Hall, 1992. 343—364.

7 Albano A, Ghelli G, Orsini R. Types for databases: the galileo experiences. Proc. of the 2nd Int. Workshop on Database Programming Language, 1990.

8 Matthes F, Sohmidt J W. The type system of DBPL. Proc. of the 2nd Int. Workshop on Database Programming Languages, 1990.

9 Jacobs D. A type system for algebraic database programming languages. Proc. of the 2nd Int. Workshop on Database Programming Languages, 1990.

10 Ohori A, Buneman P, Tannen V B. Database programming in MACHIAVELLI—a polymorphic language with static type inference. Proc. of the 2nd Int. Workshop on Database Programming Languages, 1990.

11 Fegaras L, Sheard T, Stemple D. The ADABTPL type system. Proc. of the 2nd Int. Workshop on Database Programming Languages, 1990.

12 Dearle A *et al*. NAPIER 88—A database programming language? Proc. of the 2nd Int. Workshop on Database Programming Languages, 1990.

13 Kim W. Introduction to object—oriented databases. MIT Press, 1990.

14 Tan L, Katayana T. Meta operations for type management in object—oriented database—a lazy mechanism for schema evaluation. In: Kim W *et al* ed. Deductive and Object—Oriented Databases, North—Holland, 1990. 241—258.

15 Carey M J, Dewitt D J, Vandenberg S L. A data model and query language for EXODUS. Proc. ACM—SIGMOD Conf. on Management of Data, 1988.

16 Agrawal R, Gehani N H. Rationale for the design of persistence and query processing facilities in the database programming language O++. Proc. of the 2nd Int. Workshop on Database Programming Languages, 1990.

17 Tannen V B, Buneman P, Ohori A. Can object-oriented database be statically typed? Proc. of the 2nd Int. Workshop on Database Programming Languages, 1990.

18 Cardelli L, Wegner P. On understanding types, data abstraction and polymorphism. ACM Computing Surveys, 1985, **17(4)**: 471-522.

19 Khoshafian S N, Copeland G P. Object identity. OOPSLA'86 Proceedings, 1986.

20 陈睿. 对象持久性的研究[博士论文]. 西安电子科技大学, 1993.

THE META DATA MODEL BASED ON TYPE SYSTEM

Chen Rui

(*Department of Computer Science, Southern China University of Technology, Guangzhou* 510641)

Cai Xiyao

(*Software Engineering Institute, Xi'an University of Electronic Science and Technology, Xi'an* 710071)

Abstract This paper studies the relationship between data models and the type systems of programming languages. It is noted that data models can be described in terms of types and the relationships among types. The meta model which is used to describe data models is proposed and established on the PCT type language, which is developed by combining the C++ type system and the first order predicate calculus. Various aspects of data models can be successfully described in PCT.

Key words Data model, object-oriented programming, data type, first order predicate calculus.