객체지향적 접근에 의한 프로그래밍 언어의 지속성 모델

(A Model of Object-Oriented Persistent Programming Language)

서울대학교 계산통계학과 전산과학 전공 조은선
서울대학교 컴퓨터공학과 김형주
서울대학교 계산통계학과 한상영

요약

지속적인(persistent) 데이타를 높은 수준의 표현력을 가지고 다루기 위해서는 데이타베이스와 프로그래밍 언어와의 결합이 필수적이나, 이러한 결합은 두 모델간의 불일치(impedence mismatch)를 야기시킨다. 최근에는 객체지향 개념의 도입으로, 객체지향 데이타베이스와 객체지향 프로그래밍 언어를 하나의 개념 아래 결합하게 되어 불일치가 많이 해소되었으나, 아직도 상당량의 불일치가 잔재한다. 본 논문에서는 기존의 객체지향 프로그래밍 언어 안에 확장될 수 있는 새로운 지속적 기억 장소 구조(일명 'P-구조')와, 시스템 클래스 'FUNCTION' 및, 클래스에 독립적인 집합 연산 방식을 대수적으로 제안함으로써, 궁극적으로 데이타베이스 시스템과의 모델의 불일치(impedance-mismatch)를 극복하는 방안을 제시한다.

Abstract

Managing persistent data with high-expressive power requires integration of database and programming languages. However, such integration has a generic problem, namely "impedence mismatch problem". Though recent progress of object-oriented concepts provides a foundation for seamless integration of database and programming languages under the umbrella of object-oriented database programming languages, a significant amount of impedence mismatch still remain. This paper presents a new object-oriented programming language and its model having new concepts, called "*P-structure*" and "*Operation Objects*" for better resolving the impedence mismatch problem.

1 서론

객체지향 데이타베이스와 객체지향 프로그래밍 언어의 결합은 관계형 데이타베이스와 관계형 데이타베이스 언어 사이 보다 불일치가 적다. 그 이유는 객체지향이란 하나의 개념을 공유하기 때문이다. 그러나 여기서도 불일치(impedence mismatch)의 문제가 완전히 제거된것은 아니다[4].

본 논문에서는, 객체지향 개념에 입각한, 비교적 선언적인 연산을 사용하는 지속성 모델을 제안한다. 먼저, 기존의 데이타베이스 프로그래밍 언어가 객체지향 데이타베이스에서 지원되어야할 여러 개념들이 적절히 포괄되지 못하고 있음을 지적하고, 특히 데이타베이스를 위해 가장 기본이 되는 지속성(persistence)에 대해 그 극복방안을 검토한다.

그러기 위해 우선 이러한 요구 사항을 바탕으로, 언어에 독립적으로 원하는 개념들을 추상화 시킨 후, 연관되는 언어적 도구를 마련하는 방식을 통하여, 객체지향 데이타베이스와 객체지향 언어의 모델간의 일치를 도모한다. 그리고, 집합을 하나의 연산 단위로 자유롭게 사용하며, 집합들간의 효율적인 코드 공유를 지원하기 위해, 객체지향적 관점에서 기존과는 다른 방식으로 문제를 접근한다. 이러한 언어 모델은 형식적 대수적 정의가 구성되며, C++을 기본으로한 실제 예제가 보여진다.

2 지속성 모델 - 데이타 구조

2.1 기존 언어의 클래스 모델

객체지향 데이타베이스의 클래스와 객체지향 언어의 클래스 개념 사이에는 꽤 차이가 있음을 알 수 있다[4]. 첫째, 객체지향 데이타베이스에서의 클래스는 스키마의 개념외에 객체들의 집합(set of instances)의 의미가 포함되어 있는 반면, 객체지향 언어의 클래스는 타입의 의미뿐이다. 따라서, 객체지향 언어를 사용하여 데이타베이스를 다루려면 집합에 관한 작업을 별도로 해야 한다. 한 예로, 클래스의 한 객체를 생성 할 때마다 해당 집합에 삽입 연산을 해주는 것을 들 수 있다. 여기서, 객체 생성시 자동적으로 삽입을 해주게 되면 프로그래머의 부담이 덜어지게 되는데, 객체지향 데이타베이스에서는 관계형 데이타베이스와 달리 각 집합 간의 계승 관계등을 잘 고려해야 한다. 즉, 하위 클래스에 삽입된 객체는 상위 클래스에도 자동적으로 삽입된 결과가 되어야 한다.

또, 데이타베이스에서는 객체의 생성/삭제시 클래스에로의 삽입/삭제 이외에도 집합 관련 처리가 되어야할 부분이 있다. 즉, 한 객체나 객체들의 집합이 다른 객체의 속성값이 될 수 있는데, 객체지향 데이타베이스에서는 특히, 객체들의 집합을 속성으로 갖는 경우가 많다. 의미있는 집합을 유지하려면, 여기서도 객체 하나를 생성/삭제할 때마다 다른 객체의 속성에 나타난 해당 집합에서 삽입/삭제가 수행 되어야 한다.

많은 객체지향 데이타베이스 언어에서는 사용자가 명시적으로 일정한 묶음을 설정하여 직접 삽입/삭제를 하도록 유도하는 것으로 해결한다. 예를 들어 O++[1]에서는 집합 의미의 '클러스터(cluster)'와 '서브 클러스터(subcluster)'로 이것을 지원한다. 즉, 각 지속적 클래스마다 계승 관계도 그대로 반영하는 '클러스터'를 생성할 수 있도록 하여 그 클래스에 해당하는 객체들의 집합을 나타낸다. 그리고, 프로그래머는 클래스 단위 외에도 기존의 클러스터에서 부분 집합을 취한 임의의 '서브 클러스터'를 생성, 삭제할 수 있다. 그러나, 삽입될 클러스터는 객체의 생성시에 명시되기 때문에 한 객체가 두 개 이상의 클러스터에 속하는 유연성을 지원하지 못한다.

그리고, 해당 인스턴스 집합의 표현이 클래스와 많이 다른 것도 문제점 중 하나인데, 예를 들면 E[8]나, O_2[7]등에서는 이미 기존에 존재하는 클래스에 대해서도 같은 이름의 집합을 명시적으로 생성해 주어야만 한다. 또, Gemstone/OPAL[5]에서는, 데이타베이스를 위해서는 클래스에 메세지를 주어 해당 인스턴스 집합을 그 결과로 리턴 받아 사용한다. 즉, 모두 클래스에 데이타베이스적 의미가 첨가되는 형태가 임의적이고 인위적이어서, 모델간의 불일치를 조장한다.

그리고, 특히, C나 C++을 기반으로한 데이타베이스 프로그래밍 언어들에서 for반복문의 기반이 됨으로써 연산의 단위가 될 수 있다[1][6]. 이러한 반복문을 사용하여 원소 단위의 접근을 하면 다른 대부분의 집합 단위 연산들은 이것을 바탕으로 작성될 수 있으나, 여러 클러스터들이 반복문으로 생성된 집합 단위 연산들을 공유할 수 있는 방법은 없다. 그리고, 이것은 선언적인 데이타베이스 질의와 일치하지 않는다.

살펴 본 바에 의하면, 기존의 언어가 객체지향 데이타베이스에서 지원되어야할 여러 개념들을 포괄하고 있지 못함을 알 수 있었다. 본 논문에서는 먼저 언어에 독립적으로, 원하는 개념들을 추상화 시킨 후, 연관되는 언어적 도구를 마련하는 방식으로써, 객체지향 데이타베이스와 객체지향 언어의 모델간의 일치를 도모한다. 그리고, 집합을 하나의 연산 단위로 자유롭게 사용하며, 집합들간의 효율적인 코드 공유를 지원하기 위해, 기존과는 다른 방식으로 객체지향적 관점에서 문제를 접근한다.

2.2 P-구조

앞 절까지 살펴본 바로는 프로그래밍 언어에서 데이타베이스를 주기억장치처럼 자연스럽게 다룰 수 있기 위하여 여러가지 요구되는 사항이 있었다. 이러한 사항을 다 만족시키는 적절한 도구(facility)가 되는 구조(structure)를 프로그래밍 언어의 기억 장소 모델에 삽입시키는 것이 본 논문의 목적이다. 본 절에서는, 클래스 정의에서 데이타베이스의 요구들을 포괄할 수 있도록 언어의 의미를 확장 시킨 하나의 추상적 구조 모델을 제시한다. 이러한 구조를 이하 P-구조(Persistence-Structure)라고 명명한다. 본 절과 다음절까지는 P-구조가 만족해야할 성질을 나열함으로써 다소 비형식적으로 P-구조를 정의 한다.

Property 1 *P*-구조는 스키마 클래스마다 하나씩 존재하며 이 경우는 해당 클래스의 객체가 생성, 삭제될 때마다 *P*-구조도 삽입, 삭제가 이루어진다. 해당 클래스의 계승 관계도 그대로 유지한다. 단, 임시 객체는 제외한다.

즉, 매 클래스 마다 하나의 P-구조가 존재하여 객체가 생성, 삭제될 때 해당 P-구조에 반영되는 것으로서, O++의 클러스터 접근 방법과 동일 하다. 그리고, 스키마 클래스의 모든 객체는 디폴트로 P-구조에 속하도록 한다. 지속 시간은 각 변수의 선언부나 생성부에서 결정되게 되는데, 따라서, 타입 수직성[2]이 유지되게 된다. 또, 이러한 방식은 선언시 부터 변수의 시간적인 속성이 결정되므로 컴파일 시간 오류 점검이 가능하다. 다음이 그 간단한 프로그램 예이다.

```
DDL에서
     class item{···}       // 스키마 클래스
                 //일반적인 클래스와 같은 방식의 선언
프로그램에서
     item cpu1(0.005); // 지속적인 객체
     item cpu2(0.01); // 지속적인 객체
     temporary item simulation(0.002);// 임시적인 객체
     temporary item *simulationpt;
     simulation = cpu1;
     cpu2 = simulation;
     simulationpt = new temporary item(0.002) ;
     ⋮
```

Property 2 복합 클래스마다 하나씩 존재한다. 이 경우도 해당 클래스의 객체가 생성, 삭제될 때마다 *P*-구조도 삽입, 삭제가 이루어진다.

O++에서는 스키마 클래스에 해당되지 않는 서브 클러스터들은 기존의 클러스터로부터 부분 집합만을 취하여 정의 될 수가 있다. '복합 클래스'란, 존재하는 클래스가 나타내는 집합에 부분 집합 관계 이상의 함수를 적용하여 얻어 지는 집합을 일컫는다. 그러한 함수는 다음 연산을 통해 재귀적으로 반복 적용 되는 것을 원칙으로 한다.

–클래스나 복합 클래스들의 리스트
–클래스나 복합 클래스의 행 추출연산(selection)
–클래스나 복합 클래스의 열 추출연산(projection)

이러한 복합 클래스는 다른 객체의 속성 값, 질의어 결과, 뷰등에도 대응될 수 있다. 그리고, O++에서는 한 객체가 오직 한 클러스터에 밖에는 속할 수 없지만, 여기서는 연산에 의해 객체가 선택되므로, 한 객체를 포함하는 복합 클래스는 여러개 있을 수 있다는 장점을 지닌다. 생성된 복합 클래스 간에는 위 세 가지 타입의 조합을 아크(arc)로 가지는 세멘틱 네트웍(sementic network)이 형성된다.

Property 3 내용으로 접근될 수 있다*(content addressible).*

일반적으로 데이타베이스의 정보를 추출할 때는 기반이 되는 클래스와 함께 정보의 내용의 일부(혹은 그 일부가 포함된 조건식)를 시스템에 주고 결과를 받는다. 복합 클래스에 대한 지원에서 행 추출연산이 제공되는 경우, 내용에 의한 접근이 쉽도록 그 형식을 정의한다면 데이타베이스에 보다 가까운 기억장치 구조가 될 것이다.

이와 같은 성질을 만족하는 추상적 기억장치 구조인 P-구조는 다음과 같은 문법을 가지도록 정의하였다.

DDL_type_name	::=	*string*
attr_name	::=	*string*
base_type	::=	*null* |
		DDL_type_name
P-structure	::=	(*P-structure_list*)[*select_condition*]*ret_attr*
P-structure_list	::=	*base_type* | *P-structure* |
		P-structure, *P-structure_list*
select_condition	::=	*null* |
		. | // 한 원소를 선언적으로 지칭한다.
		// SQL의 튜플 변수에 해당
		: *qexpr* :
ret_attr	::=	*null* |
		.(*attr_name_list*)
attr_name_list		*attr_name* |
		attr_name, *attr_name_list*
qexpr	::=	*op*(*qexpr_list*) |
		pathname |
		~*qexpr* |
		...

위의 구문을 사용한 간단한 예를 들면 다음과 같다.

```
class basepart{···} //DDL에서
basepart[:cost < 100:].cost += 100 ; //사용
```

basepart는 클래스 이름인 동시에 집합의 의미를 지니며, 추출 조건은 [ ] 사이에 조건식으로 넘겨 준다. 따라서, 타입 개념과 객체의 집합 개념이 공존하게 된다. 그러나, 집합을 접근하는 구문이 [ ]를 필요로 하기 때문에 구문상으로 혼동할 염려가 없다. 게다가, 위와 같은 구문을 사용하면 식별자가 아니라 값(조건)으로 곧바로 접근할 수 있어 편리하다.

2.3 연산

앞 절의 basepart[:cost < 100:].cost는 하나의 P-구조로서, 질의어의 결과나 다른 클래스의 속성으로 들어간다. 본 논문에서는 이것을 관련 연산과 함께 묶어 저장하게 되는데, 'FUNCTION'이라는 시스템 클래스를 도입하여, P-구조에 결합된 연산 집합을 나타내는, 'FUNCTION'의 인스턴스를 생성한다. 본 절에서는 이러한 연산에 대해 자세히 언급함으로써, 앞 절에서 시작한 P-구조의 정의를 확장한다.

Property 4 각 *P*-구조마다 집합 단위로 적용 가능한 연산들이 존재한다.

각 P-구조마다 고유한 연산 집합을 가지고 있도록 하는 방법은 여러 가지가 있을 수 있다. 가장 간단한 방법은 각 스키마 클래스안에 메쏘드가 정의되는 형식이다. 각 메쏘드 집합이 해당 P-구조간의 관계를 따라 세멘틱 네트웍을 형성할 수 있으며, 여기서 다른 P-구조에서 쓰였던 메쏘드의 계승으로 재사용이 가능하다. 그러나, P-구조들 내의 관계에 따라 메쏘드끼리의 관계가 지어지기 때문에 여러 유용한 재사용이 불가능한 경우가 있다. 즉, 비슷한 연산들이 계승 관계와는 무관한 클래스들 간에 공용되는 경우를 지원하지 못한다. 따라서 메쏘드 간의 관계와 해당 P-구조끼리의 관계는 서로 독립적이므로 각각은 자체적으로 다른 계승관계를 가지는 것이 바람직 하다.

이를 위하여 프로그래머는 비슷한 코드를 여러개 쓰든지, 각 클래스 이름들을 인자로 받는 매크로를 사용할 수 밖에 없었다. 그러나, 데이타베이스 언어는 선행처리기(preprocessor)에 의해 일반 프로그래밍 언어로 변환되는 방식을 주로 채택하고 있어서[1][8], 이러한 매크로를 사용할 수 있는지 여부가 각 언어의 변환 방식에 따라 결정되게 된다. C++에서 제공되는 Template[9]은 보기는 비슷하나 클래스 자체가 직접 인자로 사용되는 것을 지원하지 않는다.

본 논문에서는 한 클래스에 적용되는 연산들의 집합도 별도의 하나의 객체로 본다. 이것은 언어 자체에서, 속성은 없고, 모든 클래스에서 공유 가능한 기본적인 연산들을 메쏘드로 하는 FUNCTION이란 클래스를 제공함으로써 가능하다. 프로그래머는 초기화 인자로 하나의 P-구조를 보내어 FUNCTION의 객체를 생성할 수 있다. 그러면, FUNCTION의 객체는 소멸될 때까지 해당 P-구조에 대한 연산을 담당하게 된다. 따라서, FUNCTION내에 정의된 공용 가능한 메쏘드들은, 해당 P-구조(집합)의 단위로 적용될 수 있다. 그러므로, FUNCTION에 정의된 연산들은 FUNCTION객체 생성을 통하여 다양한 P-구조에 적용될 수 있다.

보다 복잡한 연산들은 프로그램 프로그래머가 FUNCTION을 계승하여 새로운 클래스로 생성할 수 있다. 이렇게 만들어진 새로운 연산 클래스는 FUNCTION에서 계승받은 것 외에도, 다른 여러 코드들을 포함하게 된다. 이 때, 포함시킨 코드를 다른 P-구조에서도 그대로 사용하고 싶다면, 원하는 P-구조를 그 새로운 연산 클래스에 인자로 주어 연산 객체를 하나 생성시킨다. 따라서 연산 클래스들간의 관계도 적용 범위에 따라, FUNCTION을 루트로 한 별도의 DAG구조가 되며, P-구조들 전체의 집합은 연산 클래스에 의해 분할(partition)된다. 간단히 예를 들면 다음과 같다.

```
class basepart{ ··· };
     ⋮
FUNCTION CheapPart( basepart[:cost < 100:] );
          // 해당 P-구조를 위한 연산용 객체 생성
CheapPart.update(:cost+=100:);
CheapPart.print();
```

첫번째 문장에서는 FUNCTION 클래스에 초기화 인자로 P-구조 basepart[:cost < 100:]를 보내어, 이 인자를 위한 연산 객체 CheapPart를 생성 하였다.

P-구조는 프로그래머가 자유롭게 변수를 할당할 수 있는 사용자 타입의 일종이 아니라, 단지 하나를 생성할 수 있도록 하였다. 그리고, 질의어 결과나 다른 객체의 속성 값이 되는 경우 등, 만들어 놓은 P-구조를 변수에 저장할 필요가 있을 때 바로 이 연산 객체를 저장한다. 이로써 프로그래머의 입장에서는 위 두번째 문장등에서 나타난, 해당 객체의 연산을 호출함으로써 연산이 수행되는 방식과 함께, 각 P-구조가 하나의 객체이고 그 객체마다 코드가 캡슐화 된것으로 간주되어 질 수 있다. 즉, 객체지향 데이타베이스는 보다 객체지향적 개념을 가지고 프로그래밍 언어로 삽입 되었다.

3 지속성 모델 - 프로그래밍 예

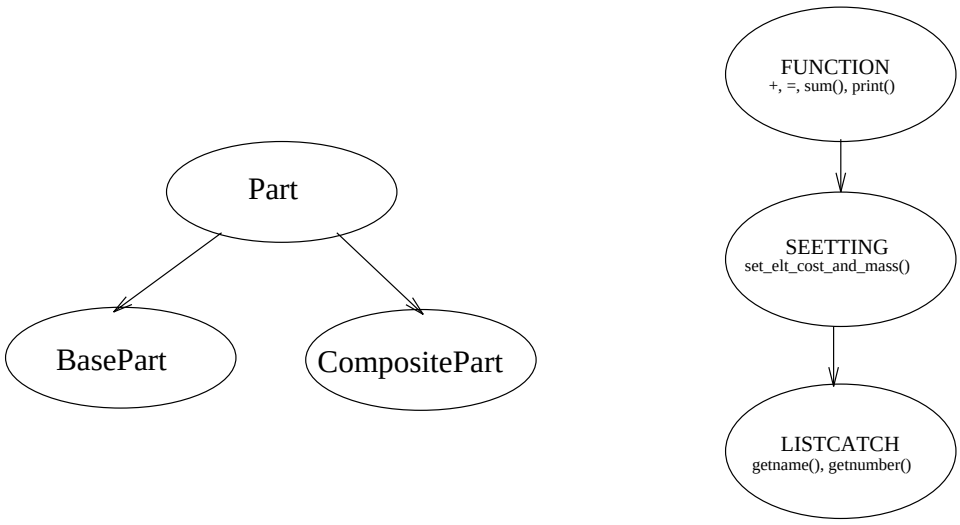
다양한 프로그래밍 언어의 계산능력(expressive power)을 측정하기 위해 Atkinson과 Bunman[2]은 네가지의 벤치마크를 제안했다. 본 장에서는 P-구조를 사용하여 이 벤치마크를 프로그램하였다.

제시된 벤치 마크는 제조 공장 부품 데이타베이스를 이용하였다. 부품은 기본 부품이 있고 그를 이용하여 만드는 복합 부품이 있다. 복합 부품 역시 다른 복합 부품을 구성하는데 사용할 수 있다. 복합 부품의 가격과 무게는 그를 구성하는 기본 부품 및 복합 부품의 합에 구성시 증분을 더하여 얻어진다.

1. **작업1(데이타베이스 기술):** 데이타베이스 스키마는 데이타베이스 정의 화일에 스키마 클래스를 정의함으로써 가능하다. 스키마 클래스는 일반 클래스와 정의 방식이 동일하고, 이렇게 정의된 정의 화일은 다른 프로그램에 삽입되게 된다.

```
class Part{ String name;
            int    number;
            static int    count;
            int    cost;
            int    mass;
            LISTCATCH supply;
            LISTCATCH subpart;
            void Part(string nm = "noname", int no = count,
                   LISTCATCH sub = Null
                   LISTCATCH supp = Null;) {
                   count++ ;
                   name = nm;
                   number = no;
                   supply = supp + LISTCATCH(
                     part[:this.number == subpart.get_number():]);
                   supply[.].subpart += this;
                   subpart = sub; } };

class Basepart:Part{
            void Basepart(string nm, int no ,LISTCATCH sub,
                   LISTCATCH supp, int ct= 0, int ms = 0)
```



<DATA HIERARCHY >

<FUCNTOIN CLASS HIERARCY>

그림 1: 데이타 클래스와 연산 집합 클래스의 계층 구조

```
                    :  Part(nm, no, sub, supp){
                         cost= ct; mass= ms; } };

class Compositepart:Part{
            int    costinc;
            int    massinc;
            void Compositepart(string nm, int no ,LISTCATCH sub,
                   LISTCATCH supp, int ctinc= 0, int msinc = 0)
                   :  Part(nm, no, sub, supp){
                     costinc= ctinc; massinc= msinc; } };

LISTCATCH : SETTING {
     String get_name(){ return(thisptr[.].name); }
     int    get_number(){ return(thisptr[.].number); } };

class SETTING : FUNCTION {
            void SETTING() : FUNCTION {
                   set_elt_cost_and_mass(thispstr[.]); }
            void set_elt_cost_and_mass(Basepart elt){
                   cost = get_cost();
                   mass = get_mass(); }
            void set_elt_cost_and_mass(Compositepart elt) {
                   set_elt_cost_and_mass(elt.subpart[.]);
                   cost = elt.subpart.sum(cost) + costinc;
                   mass = elt.subpart.sum(mass) + massinc; } };
```

'+'은 FUNCTION에서 제공해주는 연산으로 객체나 P-구조를 인자로 가지는 함수이며, 기능은 이러한 인자를 자신의 해당 P-구조에 덧붙인다는 의미로 오버로딩 되었다. sup-

ply와 subpart에 쓰인 '=' 역시 FUNCTION에서 오버로딩으로 지원된 연산자이며, SETTING에서 사용된 sum(*Attribute-Name*)도 FUNCTION에서 지원되는 함수로, SQL의 sum과 유사하게 자신의 P-구조의 원소들이 가지고있는 특정 속성을 모두 더하는 함수를 제공한다. 데이타 클래스 계층 구조와 연산 집합 클래스의 계층 구조는 그림1과 같다. cost와 mass를 구하는 set_elt_cost_and_mass()는 오버로딩에 의해 인자의 타입에 따라 각기 다른 함수가 불릴 수 있다.

2. **작업2(단순한 질의):** 두 번째 질의는 가격이 100달러 이상인 기본 부품의 이름, 가격, 무게를 출력하는 것이다.

```
SETTING lowcostpart(: Basepart[cost>100].(name,cost,mass):);
lowcostpart.print();
```

3. **작업3(재귀적 질의):** 세 번째 질의는 이름이 mast인 부품의 무게와 가격을 구한다. 여기서는 이름, 무게, 가격의 세 속성을 가지는 P-구조를 정의한다.

```
SETTING mastpart(part[: name = "mast" :].(name, cost, mass));
mastpart.print();
```

4. **작업4(새로운 객체 생성):** 네번째 질의는 새로운 복합 부품을 데이타베이스에 삽입하는 것이다. 이 때 구성되는 부품과 구성하는 부품간의 관계도 데이타베이스에 넣어야 한다.

```
LISTCATCH sup1(part[: number == 1 || number == 2 || number == 3 :]);
LISTCATCH sub1(part[: number == 12 || number == 13 :]);
CompositePart newpart("newpart", 100, 27, 30, 7, sup1, sub1);
```

한가지 주목할 사실은 스키마등 클래스 정의 부분의 비중이 실제로 프로그램 하는 부분보다 훨씬 커졌다는 것이다. 이것은 해당 클래스의 성격을 적절히 파악하여 설계하기만 하면 후에 프로그래밍과 재사용이 상당히 간단해 진다는 것을 의미한다. 그리고, 응용에 따라 데이타베이스가 복잡해지면, 스키마와 프로그램의 두 개의 단계가 아닌, 여러 단계를 둔 점증적인 프로그램이 가능하여, 데이타베이스 사용자의 수준에 따라 유용한 기능을 미리 제공할 수도 있다.

4 지속성 모델 - 대수적 정의

본 장에서는 앞서 설명된 지속성 모델에 관한 대수적 모델을 제시한다. 집합론에 기초를 두었으며, 전체 구조에 대하여 단계적으로 모델을 구성한다.

Definition 1 기본 도메인*(Base Domain)* $\mathbf{D}$란 도메인 $D_1, D_2, \cdots, D_n$ 의 유한 집합이다. 시스템에서 기본적으로 정의된 타입을 의미하며, 각 D_i간에는 교집합이 없다

Definition 2 $v \in d$ *s.t.* $d \in \mathbf{D}$인 v의 집합을 *Base Value* $\mathbf{V}$ 라 한다*[3]*.

Definition 3 레이블 집합 $\mathbf{L}$이란 스트링의 집합이다. 특별히 $ATTRs, METHs, SUP, SUB$의 네가지의 특수값을 원소로 포함하고 있다.

Definition 4 이름 집합 $\mathbf{N}$은 임의의 고유한 스트링의 집합이다. 즉, $n \in \mathbf{N}$인 어떠한 두 값도 같지않다.

객체(Object)

객체는 각 기본 도메인이나, 레이블과 객체 쌍의 순서화 리스트, 특수한 값으로 $Null$이 포함된다.

Definition 5 객체들의 집합 $\mathbf{O}$의 원소 o는 다음과 같이 정의한다.

1. 어떤 i에 대하여 $o \in D_i$ 일 때 *(*단, $D_i \in \mathbf{D}$*)*.
2. $l_i \in \mathbf{L}$ 이고 $o_i \in \mathbf{O}$일 때 $\langle l_1 : o_1,\ l_2 : o_2,\ \ldots,\ l_n : o_n \rangle$. *(*$i \in n$, $n < \infty$인 정수*)*
3. *Null*

타입(type)

Definition 6 타입의 집합 $\mathbf{T}$는 다음의 성질을 만족한다.

1. 각 D_i에 대하여, 타입 τ_i가 있어 $\mathbf{T}$에 속한다. 이 때, 원소 $e \in D_i$는 $e \in \tau_i$이다.
2. $\langle\langle l_1 : \tau_1,\ \ldots,\ l_n : \tau_n \rangle,\ \langle f_1,\ \ldots,\ f_n \rangle\rangle$에 대하여, 타입 τ가 있어 $\mathbf{T}$에 속한다.
   그리고, $o_1 \in \tau_1,\ \ldots,\ o_n \in \tau_n$이면, $\langle l_1 : o_1,\ \ldots,\ l_n : o_n \rangle \in \tau$이고, 모든 f_i에 대하여 $f_i(\langle l_1 : o_1,\ \ldots,\ l_n : o_n \rangle) = g_i(X_i)$인 g_i와 X_i가 존재한다. *(*단, $i \in n$, $n < \infty$인 정수이고, X_i는 객체들의 리스트*)*
3. *Null*

타입은 $Null$ 혹은 기본 도메인에 대응되는 것이거나, 또는 레이블로 리스트를 이룬 복합 객체와 그에 적용될 함수 리스트가 묶여진 것 세 가지 경우중 하나임을 정의하였다. 여기서 함수 f_i는 다른 부가적인 상태 변화(side effect)를 야기시키지 않는다고 가정하였다. 상태를 한 객체로 표현하고 모든 객체에 인위적으로 상태를 나타내는 레이블을 위치시키면, f_i의 정의에 의해, 상태 변화까지를 반영할 수 있는 적절한 g_i를 선택할 수 있다.

속성(attribute)

Definition 7 *ATTRIBUTE_LIST*란 $l_i \in \mathbf{L}$ 이고 $\tau_i \in \mathbf{T}$일 때, $\langle l_1 : \tau_1,\ l_2 : \tau_2,\ \ldots,\ l_n : \tau_n \rangle$ 이다. *(단, $i \in n$, $n < \infty$인 정수)*

메쏘드(Method)

Definition 8 *Method_LIST*는 특정 타입 τ에 속하는 o에 대해, $f_i(o) = g(X_i)$인 $g(X_i)$가 존재할 때, $\langle f_1,\ f_2,\ \ldots,\ f_n \rangle$ 을 말한다. *(단, $i \in n$, $n < \infty$인 정수)*

즉, $Method_LIST$란 특정 타입에 적용될 수 있는 함수들의 집합을 말한다.

클래스(Class)

본 모델에서는 4가지의 메타 클래스(meta class)를 정의한다. $CLASS$는 가장 기본이 되는 메타 클래스로, 일반적으로 프로그래밍 언어에서의 클래스와 대부분의 클래스들을 원소로 가지고 있다. $SCHEMA$는 데이타베이스의 성격을 제공하는 메타 클래스이고, $PSTR$ 메타 클래스는 $CLASS$와 $SCHEMA$에서 다중 계승 된다. 따라서 PSTR에 속하는 원소는 이 일반적인 클래스의 성격과 함께 데이타베이스에서의 집합 개념도 함께 가질 수 있다. $MFUNCTION$은, 원소로써 앞 장에서 사용하였던 클래스 FUNCTION 하나만을 가지는 메타 클래스이다. $CLASS$에서 단지 P-구조와 그를 관리하는 함수를 첨가하기 위해 계승한 것으로, 집합 개념의 P-구조를 클래스 단위와 일치시킨 결과 필요한 정의이다.

Definition 9 *CLASS*란 다음과 같은 값을 가지는 객체이다.

$\langle NAME$: "$CLASS$",
$ATTRs$: $\langle name$: **N**,
$ATTRs$: ATTRBUTE_LIST,
SUP : Class_LIST,
SUB : Class_LIST,
$METHs$: Method_LIST $\rangle$,
SUP : $\langle$Null$\rangle$,
SUB : $\langle$PSTR, MFUNCTION $\rangle$,
$METHs$: list of initializer of class, list of destructor of class $\rangle$

Definition 10 *SCHEMA*란 다음과 같은 값을 가지는 객체이다.

$\langle NAME$: "$SCHEMA$",
$ATTRs$: $\langle BASE\_CLASS$: Class_LIST,
$QUERY$: PREDICATE,
RA : ATTRIBUTE_LIST, $\rangle$,
SUP : $\langle$Null$\rangle$,
SUB : $\langle$PSTR$\rangle$,
$METHs$: Method_LIST$\rangle$

Definition 11 *PSTR*이란 다음과 같은 값을 가지는 객체이다.

$\langle NAME : \text{“}PSTR\text{”},$
$ATTRs : \langle$ `Null` $\rangle,$
$SUP :$ $\langle$ `CLASS, SCHEMA` $\rangle,$
$SUB :$ $\langle$ `Null` $\rangle,$
$METHs :$ `list of initializer of PSTR, list of destructor of PSTR` $\rangle$

Definition 12 *MFUNCTION*이란 다음과 같은 값을 가지는 객체이다.

$\langle NAME : \text{“}MFUNCTION\text{”},$
$ATTRs : \langle$ `PSTR` $\rangle,$
$SUP :$ $\langle$ `CLASS` $\rangle,$
$SUB :$ $\langle$ `Null` $\rangle,$
$METHs :$ `list of initializer of FUNCTION, list of destructor of FUNCTION` $\rangle$

관계(relation)

Definition 13 $o \in \mathbf{O}$, 일 때, *IsElementOf*는 다음을 만족하는 관계*(relation)*이다.
즉, $\langle o,\ c \rangle \in$ *IsElementOf*라는 것은 아래 모두가 성립 되어야 한다.
c에 $ATTRs$이 있어서 $\langle l_1 : c_1,\ \ldots,\ l_m : c_m \rangle$이고, c의 SUP가 있다면 그 SUP의 $ATTRs$가 반드시 존재하여 그 값이 $\langle l_1 : d_1,\ \ldots,\ l_s : d_s \rangle$이고, $o = \langle l_1 : o_1,\ \ldots,\ l_n : o_n \rangle$ 이라면,

1. 모든 c_i에 대해 $\langle c_i,\ CLASS \rangle \in$ *IsElementOf* 거나 $c_i \in \mathbf{T}$이다.

2. c에 RA가 없을 때에는 다음과 같은 다중 인자 함수 $G(c_1,\ \ldots,\ c_m)$가 존재한다.
   $G = \langle g_1,\ g_2,\ \ldots,\ g_m \rangle$ *s.t.* $g_1(c_1) = o_{k_1}, g_2(c_2) = o_{k_2}, g_m(c_m) = o_{k_m}$
   단, $k : m \longmapsto n$ 인 단사 함수*(injective function)*이다. $(n \leq m)$
   (만일 c에 RA가 있으면 RA에 속하는 원소 c_i에 대해서만 위가 성립)

3. c의 SUP가 *Null*이 아니고 SUP에 RA가 없을 때, 다음과 같은 다중 인자 함수 $H(d_1,\ \ldots,\ d_n)$가 존재한다.
   $H = \langle h_1,\ h_2,\ \ldots,\ h_s \rangle$ *s.t.* $h_1(d_1) = o_{j_1}, h_2(d_2) = o_{j_2}, h_s(d_s) = o_{j_s}$
   단, $j : s \longmapsto n$ 인 단사 함수*(injective function)*이고, 치역*(Range)*은 n에서 위 *2*에서 결정된 치역을 뺀 차집합이다. $(s+m \leq n)$
   (만일 SUP에 RA가 있으면 RA에 속하는 원소 d_i에 대해서만 위가 성립)

4. c가 $METHs$ 레이블을 가지는 경우에는, $METHs$의 모든 m에 대해서 $\langle o,\ e \rangle \in m$이 성립하는 어떤 객체 e가 존재한다.

5. c가 SUP 레이블을 가지는 경우에는 SUP안에 있는 원소들도 $METHs$ 레이블을 가지는 객체이어야 하며, $METHs$의 모든 n에 대해서 $\langle o,\ p \rangle \in n$이 성립하는 어떤 객체 p가 존재한다.

6. c에 $QUERY$ 레이블이 존재하는 경우에는 o는 해당 $QUERY$가 명시하는 조건을 만족해야만 한다.

Definition 14 *Class*는 $\langle$ *Class, CLASS* $\rangle \in IsElementOf$인 객체이다.

Definition 15 $Class_LIST$는 $\langle c_i,\ CLASS \rangle \in IsElementOf$일 때, $\langle c_1,\ c_2,\ \ldots,\ c_n \rangle$ 을 말한다. *(단, $n < \infty$인 정수)*

여기서, 다음과 같은 사실들을 얻어낼 수 있다.

Theorem 1 *CLASS, SCHEMA, PSTR, MFUNCTION*은 모두 *Class*이다.

CLASS는 ATTRs로 $\langle name, ATTRs, SUP, SUB, METHs \rangle$을 가진다. 그런데, CLASS자체도 $name, ATTRs, SUP, SUB, METHs$을 레이블로 하고, 그 구체적인 값을 가지므로, 적절한 METHs를 정의한다면 $\langle CLASS, CLASS \rangle \in IsElementOf$임을 알 수 있다. □

Theorem 2 *SCHEMA*는 *Class*이다.

앞의 증명에서도 보았듯이 CLASS는 ATTRs로 $\langle$ *name, ATTRs, SUP, SUB, METHs* $\rangle$을 가진다. 그런데, SCHEMA는 name, ATTRs, SUP, SUB, METHs을 레이블로 하고, 그 구체적인 값을 가지므로, 적절한 METHs를 정의한다면 $\langle$ SCHEMA, CLASS $\rangle \in IsElementOf$임을 알 수 있다. □

비슷하게 PSTR, MFUNCTION도 class임을 보일 수 있다. 다음 정리는 타입과 클래스와의 관계를 말해준다.

Theorem 3 타입 τ가 D_i나 *Null*이 아니라면, τ에 해당하는 *Class*를 만들 수 있다.

타입 τ가 D_i도 아니고, Null도 아니므로 τ는 $\langle\langle l_1 : \tau_1,\ \ldots,\ l_n : \tau_n \rangle,\ \langle f_1,\ \ldots,\ f_n \rangle\rangle$의 형태를 취한다. 이것이 CLASS에 IsElementOf임을 보인다. 앞의 증명에서도 보았듯이 CLASS는 ATTRs로 $\langle name, ATTRs, SUP, SUB, METHs \rangle$을 가진다. 먼저, ATTRs은 각 l_i, τ_i 쌍을 그 값으로 취한다. 그리고, METHs는 f_i 리스트를 그값으로 한다. 여기에 name을 임의의 고유 번호로, SUP, SUB를 $Null$로 한다면 IsElementOf의 정의를 만족하여, 증명은 끝난다.□

Definition 16 $\langle Class1,\ CLASS \rangle \in IsElementOf$, $\langle Class2,\ CLASS \rangle \in IsElementOf$일 때, $\langle Class1,\ Class2 \rangle \in IsA$는 다음을 모두 만족할 때 정의되는 관계*(relation)*이다.

1. *Class1*의 SUP에 *Class2*가 있고, *Class2*의 SUB에 *Class1*이 있다.

2. *Class2*의 $ATTRs$*(RA*가 있으면 *RA)*가 $\langle l_1 : \tau_1,\ \ldots, l_n : \tau_n \rangle$, *Class1*의 $ATTRs$*(RA*가 있으면 *RA)*가 $\langle q_1 : \gamma_1, \ldots, q_m : \gamma_n \rangle$ 일 때, $j : N \longmapsto M$이고, $l_i = q_{j(i)}$이고 $\gamma_{j(i)} \subseteq \tau_i$이거나 $\langle \gamma_{j(i)}, \tau_i \rangle \in IsA$인 단사함수*(injective function)* j가 존재한다. 단, $N = \{1 \ldots n\}$, $M = \{1 \ldots m\}$.

3. *Class2*의 $METHs$가 $\langle f_1, \ldots, f_n \rangle$ *Class1*의 $METHs$가 $\langle h_1, \ldots, h_m \rangle$ 일 때, $k : N \longmapsto M$이고, $f_i = h_{k(i)}$인 단사함수*(injective function)* k가 존재한다. *(*단, $N = \{1 \ldots n\}$, $M = \{1 \ldots m\}$*)*

4. $\langle f,$ *MFUNCTION* $\rangle \in$ *IsElementOf*면, f는 *FUNCTION*이거나 $\langle f,$ *FUNCTION*$\rangle \in$ *IsA*이다.

P-구조(P-structure)

이제 P-구조와 관련된 특수한 성질에 대해 정의한다.

Definition 17 $\langle P_1, PSTR \rangle \in IsElementOf$이고, $\langle P_2, PSTR \rangle \in IsElementOf$일때, $\langle p, P_1 \rangle \in IsElementOf$인 p에 대해, $\langle p, P_2 \rangle \in IsElementOf$이면, $\langle P_1, P_2 \rangle \in IsSelectOf$를 만족한다고 한다.

Definition 18 $\langle P_1, PSTR \rangle \in IsElementOf$이고, $\langle P_2, PSTR \rangle \in IsElementOf$일때, 다음을 모두 만족하면, $\langle P_1, P_2 \rangle \in IsProjectOf$의 관계가 성립한다고 한다.

1. $P_1.QUERY == P_2.QUERY$
2. $P_1.RA \supseteq P_2.RA$
3. $P_1.METHs \subseteq P_2.METHs$, 단, 만일 $m \in P_2.METHs$인 m에 대해 m이 $P_2.RA - P_1.RA$를 사용하면, $m \notin P_1.METHs$

즉, 데이타베이스에서 사용되는 각 집합 간의 관계(relation)를 SQL의 행 추출, 열 추출 연산 맞게 정의하였다.

Definition 19 $\langle P_1,$ *PSTR*$\rangle \in$ *IsElementOf*, $\langle P_2,$ *PSTR*$\rangle \in$ *IsElementOf*일 때, $\langle P_1, P_2 \rangle \in$ *IsRetrievedFrom*는 다음을 모두 만족할 때 정의되는 관계*(relation)*이다.

1. P_1의 $BASE_CLASS$를 B라 하면, $B \in P_2$이다.
2. $\langle P_1, r_1 \rangle$, $\langle r_1, r_2 \rangle$, ... $\langle r_{n-1}, r_n \rangle$ 가 *IsSelectOf*나 *IsProjectOf*에 속하는 r_1, ... r_n이 존재한다. 단, r_n 은 $r_n.BASE_CLASS \in P_2$인 가상적인 $PSTR$객체이다.
3. $P_1.METHs \subseteq P_2$ *(* 단, 만일, $m \in P_2.METHs$인 m에 대해 m이 $r_n.RA - P_2.RA$를 사용하면, $m \notin P_1.METHs$ *)*

다음은 $IsRetrievedFrom$관계가 IsA트리에 적절한 변형을 통하여 삽입될 수 있음을 보여주는 정리이다. 즉, 기존 객체지향 언어로의 효과적인 구현 가능성을 제시한다.

Theorem 4 $\langle P_1, PSTR\rangle \in IsA$, $\langle P_2, PSTR\rangle \in IsA$일 때, $\langle P_1, P_2\rangle \in IsRetrievedFrom$이면 IsA관계를 사용하여 나타낼 수 있다.

$BASE_CLASS$가 $PSTR$이 아닌 순수한 $Class$일 때, $IsProjectOf$는 IsA 트리에 해당 $BASE_CLASS$의 SUP로 하나의 새로운 $Class$를 삽입한다. $IsSelectOf$는 IsA 트리에 해당 $BASE_CLASS$의 SUB로서, $ATTRs$에 플래그가 첨가된 새로운 클래스를 생성하여 얻어진다. 플래그는 추출되었는지의 여부를 나타낸다. 만일 $BASE_CLASS$의 원소가 하나 이상이면 다중 계승을 이용하여 필요한 경우 임시적인 $Class$를 하나 삽입할 수도 있다. $BASE_CLASS$가 $PSTR$일 때는 수학적 귀납법을 이용하여 위와 같은 과정을 거친다.

그 외 프로그래밍 언어를 넘어 질의어의 수준을 다루어야 하는 부분은, 현재 연구 중에 있다.

데이타베이스(database)

끝으로 데이타베이스를 정의한다. 각각의 데이타베이스는 하나의 구체적인 객체이고, 이 객체들을 $IsElementOf$로 가지는 클래스를 다음과 같이 정의한다.

Definition 20 $DATABASE$란 다음과 같은 값을 가지는 객체이다.

$\langle NAME$: "$DATABASE$",
$ATTRs$: $\langle DD : Class\_list,\ DATA : Set,\ ISA : relation,$
$IsSelectOf : relation, IsProjectOf : relation, \ldots\rangle$,
SUP : $\langle$`Null`$\rangle$,
SUB : $\langle$`Null`$\rangle$,
$METHs$: `list of initializers and, destructors of DATABASE, and others`$\rangle$

앞의 $CLASS$, $PSTR$, $SCHEMA$, $MFUNCTION$등과 마찬가지로 그 자체도 $CLASS$와 $IsElementOf$관계를 가진다. 그리고 이것은 $CLASS$의 정의를 만족하기 때문에, 구체적인 데이타베이스 객체를 $IsElementOf$관계로 가질수 있다. DD는 $class$가 될 수 있는 모든 객체, 즉 스키마를 나타내며, $DATA$는 나머지 객체들을 나타낸다. 각 객체들의 관계는 $ATTRs$의 값으로 가진다. 현재로는 SUP와 SUB가 $Null$이지만 필요한 기능이 추가된 데이타베이스 시스템을 모델링 할 경우 $DATABASE$의 새로운 서브 클래스를 생성시킬 수 있다.

5 결론 및 향후 과제

본 논문의 P-구조는 데이타베이스 시스템 구현 입장에서 시작된 여타 데이타베이스 언어와는 달리, 프로그래밍 언어 메모리 구조 모델에서부터 이론적으로 접근하여 설계되었다. 가장 많이 사용되는 객체지향 언어인 C++을 토대로 하였으며, 데이타 자체를 내용으로부터 직접 추출(content addressible)해 낼 수도 있도록 고안되었다.

그리고, 객체지향 개념을 사용한 연산자 FUNCTION 클래스는 여러 객체 집합과 클래스 단위에 직접 적용되며, 같은 연산 집합이 서로 다른 클래스(집합)에 대해 중복 적용될 수 있다. 또, C++에서 클래스 자체가 연산 단위(1^{st} Order Value)로 되지 못하는 것을 보완하며,

데이타베이스에서 사용되는 모든 집합을 같은 연산 집합을 공유하는 집합별로 그룹화 시키는 결과를 얻는다. 연산 집합도 객체이기 때문에 기존에 있는 클래스를 계승하여 생성할 수 있으므로 재사용성이 증가된다.

본 논문에서 제시한 대수적 모델은 클래스를 하나의 객체로 간주하는 방법을 이용하기 위해 메타 클래스를 도입하여 정의하였다. 그리고, 메타 클래스 간에도 계승(IsA)등의 관계가 존재함으로써, 동시성 제어, 보안 유지등 다른 데이타베이스 관련 작업들로 개념을 확장할 수 있는 가능성이 있게 된다. 궁극적으로는 데이타베이스를 하나의 클래스를 사용하여 정의 하였는데, 구체적인 데이타베이스는 이 클래스의 객체로 모델링할 수 있다.

결과적으로 객체지향 프로그래밍 언어(여기서는 C++)의 CLASS와 데이타베이스의 클래스의 의미가 그대로 유지되면서 결합되었고, 여기에 시스템 클래스 FUNCTION의 도입으로 인하여, 캡슐화와 재사용성이 증가하여, 보다 객체지향성을 갖추게 되었다.

앞으로는, 이와 같은 모델에 알맞는 선언적 질의어를 연구할 계획이며, 최근 요구되는 트랜잭션, 보안, 버젼등을 포괄할 수 있도록 확장시킬 예정이다. 구체적인 데이타베이스 프로그래밍 언어의 완전한 구현도 추후의 연구 과제로 남긴다.

참고 문헌

[1] R. Agrawal and N. H. Gehani. "Rationale for the Design of Persistency and Query Processing Facilities in the Database Programming Language O++". In *2nd Int'l Workshop on Database Programming Languages*, Portland OR, June 1989.

[2] M. P. Atkinson and O. P. Buneman. "Types and Persistence in Database Programming Languages". *ACM Comput. Surv.*, 19(2):105–190, June 1987.

[3] F. Bancilhon and P. Buneman. *Advances in Database Programming Languages.* ACM Press, 1990.

[4] T. Bloom and S. B. Zdonick. "Issues in the Design of Object-Orieanted Database Programming Languages". In *ACM OOPSLA Conference Proceedings*, pages 441–451, Oct. 1987.

[5] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, jacob Stein, E. H. Williams, and M. Williams. "The GemStone Data Management System". In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts and Applications.* ACM press, 1989.

[6] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. "The ObjectStore Database System". *Commun. ACM*, 34(10), Oct. 1991.

[7] C. Leclude and P.Richard. "The O_2 Databse Programming Language". In F. Bancilhon and C. D. P. Kanellakis, editors, *Object-Oriented Database System- The Story of* O_2. Morgan Kanfmann Publishers, Inc., 1991.

[8] J. E. Richardson, M. J. Carey, and D. T. Schuh. "The Design of the E Programming Language". Technical Report #824, Computer Science Department, University of Wisconsin-Madison, Feb. 1989.

[9] B. Stroustrup, editor. *The C++ programming language second edition.* Addison-Wesley Publishing Company, Inc., April 1991.