OQL 질의 처리기를 위한 중첩 질의 구조 제거용 전위 모듈의 설계 및 구현[1]

(Design and Implementation of an Unnesting Front-End for an OQL Query Processor)

요 약 ODMG의 표준 객체 질의어인 OQL을 비롯하여, 최근 여러 객체 질의어에서는 위치에 관계없이 select, from, where절 등 어디서나 중첩 구조를 자유롭게 허용하여 다양한 형태의 중첩 질의(nested query)를 표현할 수 있도록 하고 있다. 게다가 이러한 중첩 질의는 질의 처리기(query processor)의 성능에 중요한 영향을 끼치므로, OQL 질의 처리기에서는 다양한 형태의 중첩 질의를 최적화하는 방안을 반드시 마련하고 있어야 한다. 본 논문은 중첩 OQL 질의의 최적화 기능을 제공하는 중첩 질의 구조 제거용 전위 모듈(unnesting front-end)을 구현하였으며, 이를 이용하여 중첩 질의 처리 능력을 가진 질의 처리기를 새롭게 구현하거나 이미 존재하는 질의 처리기를 확장하여 중첩 질의 처리 기능을 추가하고자 할 때, 중첩 구조 제거(unnesting)를 위한 구현자의 구현 부담을 최소화할 수 있도록 하였으며, 이는 중첩 질의 구조 제거용 전위 모듈에서 사용하는 대수 연산자와 질의 최적화 모듈에서 사용하는 대수 연산자(algebraic operator)를 분리함으로써 가능했다.

Abstract Many object query languages including OQL(the query language proposed by the ODMG) allow query block to be nested in any clause: select clause, from clause and where clause. The processing of nested queries can affect the performance of its query processor. Therefore, an OQL query processor should have effective optimizing techniques for nested queries. This paper proposes a new framework of an unnesting front-end for an OQL query processor. This unnesting module can minimize implementation overhead when developing a new OQL processor or extending an existing query processor to be equipped with an unnesting facility. This is accomplished by separation between logical algebra operators used in an unnesting front-end and ones in a query optimizer.

1 서론

그동안 객체지향 데이타베이스(object-oriented database)가 많은 장점에도 불구하고, 널리 보급되지 못한 것은 표준 객체 데이타 모델(object data model)과 표준 객체 질의어(object query language)의 부

[1]이 연구는 과학 기술처의 국가 지리 정보 시스템 기술 개발 사업에서 지원 받았음.

재에 기인한 바가 크다고 하겠다. 하지만, 최근 ODMG(Object Database Management Group)에서 객체지향 데이타베이스 표준으로 ODMG 2.0[1]을 새롭게 발표하여, 이러한 문제점은 점차 해결되가는 추세에 있다. ODMG 2.0에서는 표준 객체 질의어로 OQL[2]을 제공하는데 OQL은 기존 관계형 데이타베이스(relational database)의 SQL과 문법 구조가 매우 유사한 반면에, 다음 두가지 측면에서 큰 차이점을 가지고 있다. 첫째, OQL은 직교 언어(orthogonal language)로서, 질의내 위치에 상관없이 자유롭게 질의를 중첩(nesting)시킬 수 있다. 물론 SQL에서도 중첩 질의를 지원하지만, where절에서만 중첩 구조를 허용하여 OQL에 비해 다소 제한적인 형태의 중첩 질의만이 가능하다. 둘째, OQL은 다양한 데이타 타입을 지원한다. 기존 SQL에서 기본적으로 지원하는, 집합(set)과 백(bag) 뿐만 아니라, 리스트(list)와 배열(array)을 지원한다. 따라서, 일반 SQL 질의 처리기를 설계할 때와는 달리, OQL 질의 처리기를 설계할 때는, 이 두가지 차이점을 충분히 고려해야 하며, 특히 중첩 질의는 질의 처리기의 성능에 중요한 영향을 미치므로[2], 단순히 중첩 질의의 지원 여부를 떠나서, 중첩 구조 제거를 통한 중첩 질의의 최적화에 관심을 기울여야 한다.

1.1 관련 연구

중첩 구조의 제거를 통한 중첩 질의의 최적화에 관해서는 그동안 많은 연구가 이루어져 왔으며, 중첩 구조 제거 방식에 따라 이들 연구를 크게 세가지로 분류할 수 있다. 첫번째는 질의어 수준에서의 중첩 구조 제거 방식(query language level unnesting)으로 과거 SQL에서 많이 연구되었던 방식이다. 이 방식에서는 SQL의 중첩 질의를 조인 질의(join query)로 변환하는 알고리듬을 사용하여, 중첩 질의를 최적화하고자 하였고[2], 이 변환 과정에서 발생하는 일명 COUNT 버그[3]를 해결하기 위해 외부 조인(outer join)을 도입하였으며[4], 이후에도 좀더 일반화된 형태의 중첩 SQL 질의를 최적화하기 위한 많은 연구가 있었다[5, 6, 7]. 이렇듯, SQL에서는 질의어 수준에서 변환 알고리듬을 적용하여 중첩 구조를 제거하고, 이 질의를 기존의 질의 처리기로 처리하는 방식을 취하였다. 하지만, OQL을 포함한 여러 객체 질의어에서는 중첩 구조가 where절 뿐만 아니라, select절과 from절에도 나타날 수 있어, 그 처리가 훨씬 복잡하다. 따라서, SQL에서 연구된 기술들을 사용하여 객체 질의어의 일반적인 중첩 구조를 처리하는데는 많은 어려움이 따른다.

두번째 방식은 대수 수준 중첩 구조 제거 방식(algebra level unnesting)으로 객체 질의어에서의 중첩 질의를 처리하는 가장 일반적인 방법이다. [8, 9]에서는 중첩 객체 질의를 중첩 대수 연산자(nested algebraic operator)를 사용하여 대수식으로 변환하고, 여기서 나타나는 중첩 구조를 조인 연산(join operator)으로 바꿔 중첩 구조를 제거했으나, 여기에 사용되는 대수 연산자들이 상당히 복잡하고, 처리할 수 있는 중첩 질의의 종류가 일부 제한되어 있다.

마지막으로 해석 수준 중첩 구조 제거 방식(calculus level unnesting)은 질의 대수(query algebra)

[2] 앞으로 OQL은 ODMG 2.0의 OQL을 지칭하기로 한다.

수준에서 이뤄졌던 기존의 중첩 구조 제거 방식과는 달리 질의 해석(query calculus) 수준에서 대부분의 중첩 구조가 제거된다. 최근에 제안된 모노이드 컴프리핸션 해석(monoid comprehension calculus)[10, 11]을 사용하면, OQL에서 지원하는 여러 문법 구조와 데이타 타입들을 일관되게 다룰 수 있으며, OQL의 다양한 데이타 타입과 한정자(quantifier) 및 집계 함수(aggregates function)들을 모두 모노이드 컴프리핸션으로 변환하여 처리할 수 있다는 장점이 있다. 모노이드 컴프리핸션 해석을 이용한 해석 수준 중첩 구조 제거 방식은 대수 수준에서의 중첩 구조 제거 방식보다 변환 규칙(transformation rule)이 간단하여 구현이 용이하며, 논리 대수식(logical algebra expression)으로의 변환도 수월하다. 모노이드 컴프리핸션(monoid comprehension)으로 변환된 OQL 질의는 정규화 규칙(normalization rule)을 적용하여 중첩 구조가 제거되며, 이 과정에서 제거되지 못한 중첩 구조는 논리 대수 변환(logical algebra transformation) 과정에서 처리된다.

본 논문에서는 모노이드 컴프리핸션 해석을 이용한 해석 수준 중첩 구조 제거 방식을 사용하여 OQL 질의 처리기의 중첩 질의 처리 기능을 전담하는 별도의 모듈을 설계하고 구현하였다. **중첩 구조 제거용 전위 모듈**이라 불리는 이 중첩 구조 제거 시스템은 다음 두가지 용도로 사용될 수 있다. 첫째, OQL 질의 처리기를 새롭게 구현할 때, 중첩 구조 제거 기능이 구현되어 있는 본 모듈을 사용함으로써 구현자의 구현 노력을 최소화할 수 있다. 둘째, 기존에 구현되어 있는 질의 처리기를 확장하여 중첩 구조 제거 기능을 추가하고자 하는 경우, 기존 질의 처리기의 변경을 최소화하여, 구현자의 부담을 줄여준다. 이와 같은 장점들은 중첩 구조 제거용 전위 모듈에서 사용하는 논리 대수 연산자와 질의 처리기에서 인식하는 논리 대수 연산자를 서로 분리하도록 설계되었기 때문에 가능한 것들이었다. 논문의 구성은 다음과 같다. 2장에서는 모노이드 컴프리핸션 해석을 소개하고, 3장에서는 모노이드 컴프리핸션 해석을 이용하여 중첩 질의로부터 중첩 구조를 제거하는 방법을 설명하며, 4장에서 중첩 구조 제거용 전위 모듈의 처리 과정을 설명하고, 5장에서 본 논문의 중첩 구조 제거용 전위 모듈의 적용 사례를 살펴보고, 6장에서 결론과 향후 연구 계획을 끝으로 논문을 맺는다.

2 모노이드 컴프리핸션 해석(monoid comprehension calculus)

ODMG 데이타 모델을 비롯한 여러 객체 데이타 모델과, OQL등의 최근 객체 질의어에서 제공하는 기능들은 집합 이론(set theory)에 기반한 관계형 이론(relational theory)으로는 제대로 지원하기가 힘들어지고 있다. 예를 들어, OQL에서 지원하는 집합(set), 백(bag), 리스트(list), 배열(array) 등의 다양한 콜렉션 타입들, 자유로운 중첩 구조의 지원, 메쏘드의 지원 등이 있다.

이에 대한 해결책으로 제시된 것이 모노이드 해석(monoid calculus)[10]이다. 여기서는 모든 콜렉션 타입을 모노이드(monoid)라는 대수 구조로 이해함으로써, OQL에서 제공되는 다양한 타입들을 일관되게 다룰 수 있으며, 또한 모노이드 해석을 모노이드 컴프리핸션이라는 수학적 표기법으로 표현함으

로써, OQL이 가지는 직교 언어(orthogonal language)로써의 특성을 잘 반영하고 있다.

2.1 모노이드(monoid)

모노이드 $\mathcal{M}$(monoid, $\mathcal{M}$)은 항등원(identity, $zero^{\mathcal{M}}$), 단위 함수(unit function, $unit^{\mathcal{M}}$), 병합 함수(merge function, $merge^{\mathcal{M}}$)를 구성요소로 하는 대수학적인 구조로서, ($zero^{\mathcal{M}}$, $unit^{\mathcal{M}}$, $merge^{\mathcal{M}}$)으로 표기한다. 여기서, 단위 함수 $unit^{\mathcal{M}}$은 최소 단위의 객체를 생성하는 함수로 콜렉션의 경우 원소 하나로 구성된 콜렉션을 생성하는 함수로 볼 수 있다. 병합 함수 $merge^{\mathcal{M}}$은 $zero^{\mathcal{M}}$을 항등원으로 하는 연산자이다. 예를 들어, 집합의 경우 $merge^{\mathcal{M}}$함수는 합집합(union) 연산에 해당하며, $zero^{\mathcal{M}}$은 공집합(empty set)에 해당하므로, 집합은 ($\{\}$, $\{a\}$, $\cup$)으로 표현할 수 있다. 모노이드에는 두가지 종류가 있는데, $unit^{\mathcal{M}}$이 콜렉션을 생성하는 함수인 경우에는 해당 모노이드를 콜렉션 모노이드(collection monoid)라 하고, $unit^{\mathcal{M}}$이 항등 함수(identity function)인 경우에는 단순 모노이드(primitive monoid)라고 한다. 정수 1, 2, 3을 원소로 가지는 정수 집합은 모노이드를 구성하는 함수들을 사용하여 다음과 같이 구성할 수 있다.

$$merge^{set}(unit^{set}(1), merge^{set}(unit^{set}(2), unit^{set}(3)) = \{1\} \cup (\{2\} \cup \{3\}) = \{1, 2, 3\}$$

2.2 모노이드 컴프리핸션(monoid comprehension)

모노이드 $\mathcal{M}$에 대한 모노이드 컴프리핸션은 $\mathcal{M}\{\ h \mid q_1, ..., q_n\ \}$의 형태로 표기한다. 여기서, h는 컴프리핸션의 머리부(head)라고 하며, $q_1, ..., q_n$ 부분을 조건부라고 한다. 머리부는 조건부에서 명시된 조건들을 만족하는 객체들을 결과 집합에 포함시킬 때 적용되는 일종의 생성자의 역할을 한다. 조건부는 크게 다음 두종류의 조건자(qualifier)로 구성된다.

- 변수 연결자(generator): $v \leftarrow E$의 형태를 가지고 있으며, 수식 E가 포함하는 객체들 각각을 변수 v에 바인딩시키는 역할을 한다.
- 조건 한정자(filter predicate): 변수 연결자에서 실제 값과 바인딩된 변수에 대해서 조건식을 적용하여 참인지 거짓인지를 판별한다.

집합 모노이드에 대한 모노이드 컴프리핸션의 한 예를 들어보면 다음과 같다.

$$set\{\ p.name \mid p \leftarrow Students, p.age < 20\ \}$$

여기서, $Students$를 학생 객체들의 집합이라고 한다면, 위 예제는 학생들 각각을 변수 p에 바인딩시켜, $p.age < 20$을 만족하는 객체들을 결과 집합으로 뽑아내고, 이 때 $p.name$이라는 수식을 적용하여 결과 집합에는 20세 미만의 대학생들의 이름이 포함되도록 한다. 이와 같이 모노이드 컴프리핸션을 이용하

여, 조건부에 명시된 조건들을 만족하는 객체들로 구성된 집합을 표현할 수 있으며, 모노이드 컴프리헨션 해석에서 질의는 모노이드 컴프리헨션으로 표현되게 된다.

2.3 모노이드 컴프리헨션 해석과 OQL 질의

모노이드 컴프리헨션 해석은 앞에서 설명한 모노이드라는 대수적인 구조를 바탕으로, 아래 열거된 바와 같이 모노이드의 구성 요소들과 모노이드 컴프리헨션으로 구성된다.

- v: 변수
- c: 상수
- $e.A$: 프로젝션(projection)
- $< A_1 = e_1, ..., A_n = e_n >$: 레코드 생성자
- $e_1 \otimes e_2$: $\otimes \in \{ =, <, >, \leq, \geq, \neq \}$
- $zero^{\mathcal{M}}$
- $unit^{\mathcal{M}}$
- $merge^{\mathcal{M}}(e_1, e_2)$
- $\mathcal{M}\{ e \mid q_1, ..., q_n \}$: 컴프리헨션

이렇게 정의된 모노이드 컴프리헨션 해석은 OQL 질의에서 제공하는 문법 구조를 잘 반영하고 있기 때문에, OQL 질의를 모노이드 컴프리헨션 해석으로 변환하기가 상당히 수월하다. 일반적으로 select-from-where 구조를 지닌 OQL 질의는 다음과 같이 모노이드 컴프리헨션 해석을 사용하여 모노이드 컴프리헨션으로 변환할 수 있다.

$$\texttt{select } E_{result}$$
$$\texttt{from } x_1 \texttt{ in } E_1, ..., x_n \texttt{ in } E_n$$
$$\texttt{where } E_{pred}$$

$$\longrightarrow bag\{ E_{result} \mid x_1 \leftarrow E_1, ..., x_n \leftarrow E_n, E_{pred} \}$$

OQL 질의의 select절은 모노이드 컴프리헨션의 머리부에 해당하며, from절은 모노이드 컴프리헨션의 변수 연결자, where절은 모노이드 컴프리헨션의 조건 한정자에 해당한다. 구체적인 OQL 질의를 모노이드 컴프리헨션으로 변환한 예는 다음과 같다.

$$\texttt{select } e.name$$
$$\texttt{from } e \texttt{ in } Employees$$
$$\texttt{where } e.salary > 200;$$

$$\longrightarrow bag\{ \underbrace{e.name}_{\text{머리부}} \mid \underbrace{e \leftarrow Employees,}_{\text{변수 연결자}} \underbrace{e.salary > 200}_{\text{조건 한정자}} \}$$

OQL을 모노이드 컴프리헨션으로 바꾸는 것에 관한 보다 자세한 내용은 부록을 참고하기 바란다.

$$\mathcal{M}\{\ e \mid \bar{q}, v \leftarrow zero^{\mathcal{N}}, \bar{s}\ \} \longrightarrow zero^{\mathcal{M}} \quad \text{(규칙 1)}$$

$$\mathcal{M}\{\ e \mid \bar{q}, v \leftarrow unit^{\mathcal{N}}, \bar{s}\ \} \longrightarrow \mathcal{M}\{\ e \mid \bar{q}, v \equiv e', \bar{s}\ \} \quad \text{(규칙 2)}$$

$$\mathcal{M}\{\ e \mid \bar{q}, v \leftarrow merge^{\mathcal{N}}(e_1, e_2), \bar{s}\ \} \longrightarrow merge^{\mathcal{M}}(\mathcal{M}\{\ e \mid \bar{q}, v \leftarrow e_1, \bar{s}\ \}, \mathcal{M}\{\ e \mid \bar{q}, v \leftarrow e_2, \bar{s}\ \}) \quad \text{(규칙 3)}$$

$$\langle\ A_1 = e_1, ..., A_n = e_n\ \rangle.A_i \longrightarrow e_i \quad \text{(규칙 4)}$$

$$\mathcal{M}\{\ e \mid \bar{q}, v \leftarrow \mathcal{N}\{\ e' \mid \bar{r}\ \}, \bar{s}\ \} \longrightarrow \mathcal{M}\{\ e \mid \bar{q}, \bar{r}, v \equiv e', \bar{s}\ \} \quad \text{(규칙 5)}$$

$$\mathcal{M}\{\ e \mid \bar{q}, some\{\ pred \mid \bar{r}\ \}, \bar{s}\ \} \longrightarrow \mathcal{M}\{\ e \mid \bar{q}, \bar{r}, pred, \bar{s}\ \} \quad \text{(규칙 6)}$$

그림 1: 정규화 알고리듬(normalization algorithm)

3 중첩 구조 제거

앞에서 살펴본 것 이외에도 모노이드 컴프리핸션 해석은 OQL에서 가능한 여러가지 형태의 질의를 자연스럽고 효과적으로 표현할 수 있다. 따라서, 사용자가 입력한 OQL 질의는 일단 모노이드 컴프리핸션 해석으로 바뀌어 내부 형태로 존재하며, 그 다음은 정규화(normalization) 과정을 통해 보다 간단하고 최적화된 형태의 모노이드 컴프리핸션 해석을 얻게 된다. OQL에서 나타나는 중첩 질의의 일부는 이 과정을 통해 중첩 구조가 제거되며, 이 과정에서 제거되지 않는 중첩 구조는 뒤에 설명할 대수 변환 과정에서 제거된다.

3.1 모노이드 해석의 정규화

모노이드 해석은 정규화 알고리듬(normalization algorithm)이라 불리는 변환 규칙을 사용하여 정규화된 형태(canonical form)로 바뀌게 된다. 그림 1에 정규화 알고리듬을 설명하고 있다.

그림 1에 열거된 정규화 규칙들 중에서, 특히 규칙 5는 중첩 질의를 모노이드 컴프리핸션으로 표현하는 과정에서 얻어지는 중첩 컴프리핸션에 적용되는 규칙으로 이를 통해 중첩 질의의 중첩 구조를 제거할 수 있다.

다음은 정규화 알고리듬 규칙 5를 설명하기 위한 중첩 OQL 질의이다.

$$
\begin{aligned}
&\texttt{select distinct } h.address \\
&\texttt{from } hs \texttt{ in } (\texttt{select } c.hotels \\
&\qquad\qquad \texttt{from } c \texttt{ in } Cities \\
&\qquad\qquad \texttt{where } c.name \ = \ \text{``}Seoul\text{''}), \\
&\qquad h \texttt{ in } hs \\
&\texttt{where } h.name = \text{``}Hilton\text{''};
\end{aligned}
$$

위 질의는 from절에 중첩 질의를 가지고 있으며, 모노이드 컴프리헨션으로 변환한 뒤 정규화 알고리듬의 규칙 5를 적용하면 다음과 같은 결과를 얻을 수 있다.

$$
\begin{aligned}
& set\{\ h.address \mid hs \leftarrow bag\{\ c.hotels \mid c \leftarrow Cities, c.name = \text{``}Seoul\text{''}\ \}, \\
& \qquad\qquad h \leftarrow hs, h.name = \text{``}Hilton\text{''}\ \} \\
\longrightarrow\ & set\{\ h.address \mid c \leftarrow Cities, c.name = \text{``}Seoul\text{''}, \\
& \qquad\qquad h \leftarrow c.hotels, h.name = \text{``}Hilton\text{''}\ \}
\end{aligned}
$$

정규화된 모노이드 컴프리헨션에 해당하는 OQL 질의는 다음과 같으며, 결과적으로 정규화 알고리듬 규칙 5를 적용함으로써 OQL 질의에 내포된 중첩 구조를 제거할 수 있다.

$$
\begin{aligned}
&\texttt{select distinct } h.address \\
&\texttt{from } c \texttt{ in } Cities, h \texttt{ in } c.hotels \\
&\texttt{where } c.name = \text{``}Seoul\text{''}, h.name = \text{``}Hilton\text{''}
\end{aligned}
$$

4 OQL 질의 처리기

기존 질의 처리기에서의 질의 처리 과정은 다소 차이는 있겠으나, 대체적으로 그림 2의 과정을 거치게 된다. 사용자가 입력한 질의로부터 파서가 파스 트리(parse tree)를 생성한 뒤, 메타 데이타를 이용하여 타입 검사를 수행하여 질의가 적법한 것인지를 확인한 다음, 대수 변환 과정을 거치면서 논리 대수식(logical algebra expression)을 생성하게 된다. 이를 질의 최적화기로 넘겨 물리 대수식(physical algebra expression)으로 변환한 뒤 최적화된 질의 수행 계획(query execution plan)을 생성하여, 질의 수행기를 거쳐 해당 질의 결과를 산출하게 된다.

그림 2와 같은 과정을 거치는 기존의 질의 처리기에서 중첩 질의를 처리하는 방법은 크게 두가지가 있다. 우선 과거 중첩 SQL 질의를 처리하기 위해 사용되던 방법으로 질의어 소스 수준에서 변환 규칙

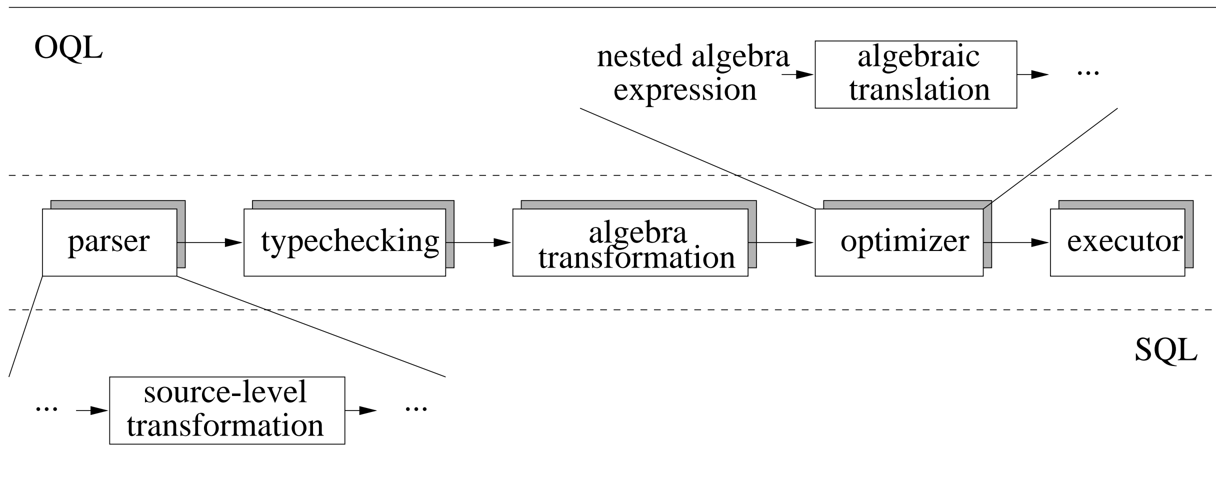

그림 2: 기존 질의 처리기에서의 질의 처리 과정 및 중첩 질의 처리

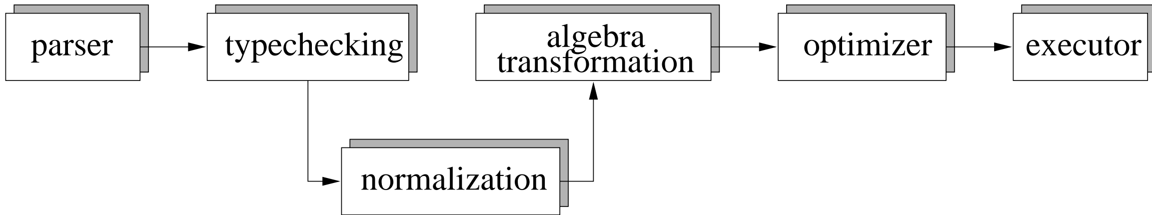

그림 3: 모노이드 컴프리헨션 해석을 적용한 OQL 질의 처리 과정

을 적용하여 중첩 구조를 제거하고, 이렇게 중첩 구조가 제거된 질의는 기존 질의 처리기의 변경없이 곧장 처리가 가능하다. 하지만, SQL에서 사용되던 이러한 방식으로는 OQL에서 지원하는 다양한 형태의 중첩 질의 구조를 지원하기가 힘들다.

두번째는 객체 질의를 처리하는 대부분의 질의 처리기가 그러하듯이, 중첩 대수 연산자들을 사용하여 중첩 구조를 지닌 질의를 파싱과 타입 검사, 대수 변환 과정을 거치면서 내부 형태(internal representation)로 표현된다. 질의 최적화 모듈에서는 중첩 대수식(nested algebra expression)을 입력으로 받아, 중첩 대수식에 포함된 중첩 구조를 제거한다. 하지만, 전반적으로 중첩 구조를 제거하는데 사용되는 변환 규칙이 너무 복잡하여 구현이 상당히 복잡하다. 그리고, 기존에 구현되어 있는 질의 처리기에 중첩 구조 제거 기능을 새로 추가하기 위해서는 이미 정의되어 있는 대수 연산자들의 구조에 많은 변경을 가져 오게 되며, 그림 2에서 보면 알 수 있듯이 질의 최적화 모듈에 많은 수정을 필요로 하게 되어 질의 처리기의 확장이 사실상 힘들어지게 된다.

그림 3은 모노이드 컴프리헨션 해석을 적용한 OQL 질의 처리기의 질의 처리 과정이다. 모노이드 컴프리헨션 해석을 적용하면 일반적으로 다음과 같은 질의 처리 과정을 거치게 된다.

1. **파싱**: OQL 질의를 파싱하여 모노이드 해석을 반영한 내부 표현으로 변환한다. 이 때, 모노이드

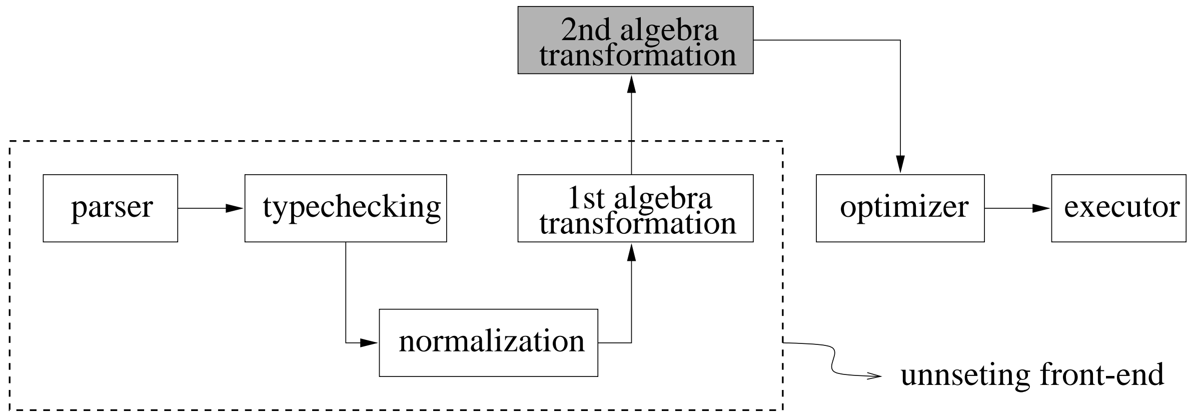

그림 4: 중첩 구조 제거용 전위 모듈(unnesting front-end)

의 종류를 알아낼 수 없는 부분이 발생할 수 있다. OQL을 모노이드 해석으로 변환하는 것에 관해서는 부록에 자세히 설명되어 있다.

2. **타입 검사**: 모노이드 해석에 대해 타입 검사를 수행하고, 전단계에서 모노이드의 종류를 알아내지 못한 것이 있었다면, 이 과정에서 해결된다.

3. **정규화**: 질의의 적법성 여부가 확인이 되면, 그림 1에 열거된 정규화 알고리듬 규칙들을 적용하여 모노이드 해석을 정규화한다.

4. **논리 대수 변환**: 정규화된 모노이드 해석으로부터 중첩 대수 연산자를 사용하여 논리 대수식으로 변환한다.

5. **최적화 및 질의**: 논리 대수식을 물리 대수식으로 변환한 뒤 이중 가장 비용이 싼 것을 선택하여 수행한다.

4.1 중첩 구조 제거용 전위 모듈(unnesting front-end)

그림 2의 처리 과정을 가지는 구조는 최적화기 앞쪽에 위치한 모듈들이 모두 논리 대수 연산자에 기반을 두고 있어, 그림 2의 질의 처리기를 그림 3과 같은 구조로 확장하기 위해서는 최적화 모듈에 상당한 수정이 불가피하다. 이러한 단점을 극복하기 위해 중첩 구조 제거용 전위 모듈(unnesting front-end, 그림 4)에서는 모노이드 정규화와 논리 대수식 생성 모듈 사이에 중간 논리 대수식(intermediate logical algebraic expression) 생성 모듈을 첨가함으로써 중첩 구조 제거용 전위 모듈에서 사용되는 논리 대수와, 최적화 모듈에서 사용하는 논리 대수간의 의존성을 제거하여 최적화 모듈에 대한 수정을 최소화한다.

다음은 중첩 구조 제거용 전위 모듈을 채택한 질의 처리기의 처리 과정이다.

1. **파싱 및 모노이드 컴프리핸션 해석으로의 변환**: OQL 질의를 파싱하여 모노이드 컴프리핸션 해석을 반영한 파스 트리를 구성한다. 여기서, 모노이드의 종류를 결정할 수 없는 부분이 있을 수 있는데, 이는 타입 검사 과정에서 해결된다.

2. **타입 검사 및 모노이드 컴프리핸션 해석의 보완**: 모노이드 컴프리핸션 해석으로 표현된 파스 트리에 대해 타입 검사를 수행하고, 전단계에서 종류를 결정하지 못한 모노이드가 있었다면 이 과정에서 해결한다.

3. **정규화**: 이 과정에서는 타입 검사를 마친 모노이드 컴프리핸션 해석에 대해 모노이드 컴프리핸션 정규화와 조건식 정규화를 반복적으로 수행하여 정규화된 형태의 모노이드 컴프리핸션 해석을 만든다.

   (1) **모노이드 컴프리핸션 정규화(monoid comprehension normalization)**: 이 과정에서는 타입 검사를 마친 모노이드 컴프리핸션 해석에 대해 그림 1의 정규화 알고리듬을 적용하여 정규화된 형태의 모노이드 컴프리핸션 해석을 만든다.

   (2) **조건식 정규화(predicate normalization)**: 이 과정에서는 모노이드 컴프리핸션의 조건 한정자에 위치한 여러 조건식들을 DNF(disjunctive normal form) 형태로 변환한다. 이 과정에서 OR는 모노이드의 $merge^{\mathcal{M}}$을 통해 다음과 같이 합집합 연산으로 바뀌게 된다.

$$\mathcal{M}\{\ e_{head} \mid e_{generators}, e_{pred1}\ or\ e_{pred2}\ \}$$
$$\longrightarrow\ merge^{\mathcal{M}}(\mathcal{M}\{\ e_{head} \mid e_{generators}, e_{pred1}\ \}, \mathcal{M}\{\ e_{head} \mid e_{generators}, e_{pred2}\ \})$$

4. **논리 대수 변환**: 여기서는 그림 3에서와는 달리 논리 대수 변환에서 한단계를 더 거치게 된다.

   (1) **1차 논리 대수 변환**: 정규화된 모노이드 컴프리핸션 해석을 중간 논리 대수 연산자(intermediate logical algebraic operator)를 사용하여 중간 논리 대수식을 생성한다.

   (2) **2차 논리 대수 변환**: 중간 논리 대수식으로부터 실제 논리 대수 연산자를 사용하여 최종 논리 대수식을 생성한다.

5. **최적화 및 실행**: 2차 논리 대수 변환 과정을 거쳐 생성한 최종 논리 대수식을 입력으로 받아, 물리 대수식을 생성한 뒤 비용을 계산하여 최적의 질의 계획을 채택하고 이를 실행한다.

4.2 1차 논리 대수 변환과 2차 논리 대수 변환

중첩 구조 제거용 전위 모듈과 질의 최적화 모듈간의 의존성을 최소화하기 위해서는 중간 논리 대수 연산자와 실제 논리 대수 연산자간의 변환 규칙을 정해 주어야 하며, 중간 논리 대수 연산자는 확장하고자 하는 질의 처리기가 제공하는 논리 대수 연산자들을 두루 포괄할 수 있도록 설계하는 것이 바람직하다.

중첩 구조 제거용 전위 모듈이 1차 논리 대수 변환에서 사용하는 중간 논리 대수 연산자들의 종류와 각 연산자들의 의미는 다음과 같으며, 이는 [12]을 참조하였다.

- **get**: 1차 논리 대수 변환이 가장 처음 적용되는 컴프리핸션[3]에서 변수 연결자(generator)의 도메인(domain) 부분이 익스텐트(extent)인 경우에 사용한다.
- **join**: 정규화된 컴프리핸션의 가장 왼쪽에 위치한 변수 연결자(generator)의 도메인(domain) 부분이 익스텐트(extent)인 경우 사용한다.
- **reduce**: 1차 논리 대수 변환의 맨 마지막 과정에서 생성되는 연산자로 컴프리핸션의 머리부(head)를 변환할 때 사용한다.
- **nest**: 컴프리핸션이 중첩되어 있는 경우에 이 중첩 구조를 제거하는 과정에서 사용된다. 주로 select절과 where절에서 중첩 구조를 가지고 있는 질의의 경우 이 연산자가 생성된다.
- **unnest**: 정규화된 컴프리핸션의 가장 왼쪽에 위치한 변수 연결자(generator)의 도메인(domain) 부분이 경로식(path expression)인 경우(from절에서 변수의 도메인(domain)이 경로식으로 표현된 경우)에 사용한다.

2차 논리 대수 변환은 1차 논리 대수 변환의 결과로 얻어진 중간 논리 대수식을 질의 최적화 모듈에서 인식하는 논리 대수식으로 변환해주며, 따라서 질의 최적화 모듈은 중첩 구조 제거용 전위 모듈에서 진행되는 작업에는 전혀 영향을 받지 않게 된다. 이것이 중첩 구조 제거용 전위 모듈이 지향하는 설계 목표이다.

5 적용 사례

본 논문에서 구현된 중첩 구조 제거용 전위 모듈은 국가 지리 정보 시스템(NGIS, National Geographic Information System)에서 개발 중인 공간 객체지향 데이타베이스 관리 시스템인 OMEGA(Object Management systEm for Geo-spatial Applications)의 질의 처리기[4]에 사용되었다. OMEGA 시스템의 질의 처리기는 공간 및 비공간 질의를 통합 처리하고 있으며, 이번에 중첩 질의를 처리할 수 있도록 시스템을 확장하는 과정에서 본 논문의 중첩 구조 제거용 전위 모듈을 사용하였다.

[3]중첩된 컴프리핸션의 경우, 한 컴프리핸션 안에 여러개의 컴프리핸션이 존재한다. 이 경우 각각의 컴프리핸션에 대해 1차 논리 대수 변환이 이루어진다.

[4]1998년 10월 현재, “NGIS 기술 개발 사업”의 “DB 툴(tool) 중과제” 내의 “공간 객체 관리 시스템 개발(development of spatial object management system) 세부 과제”에서 개발 중

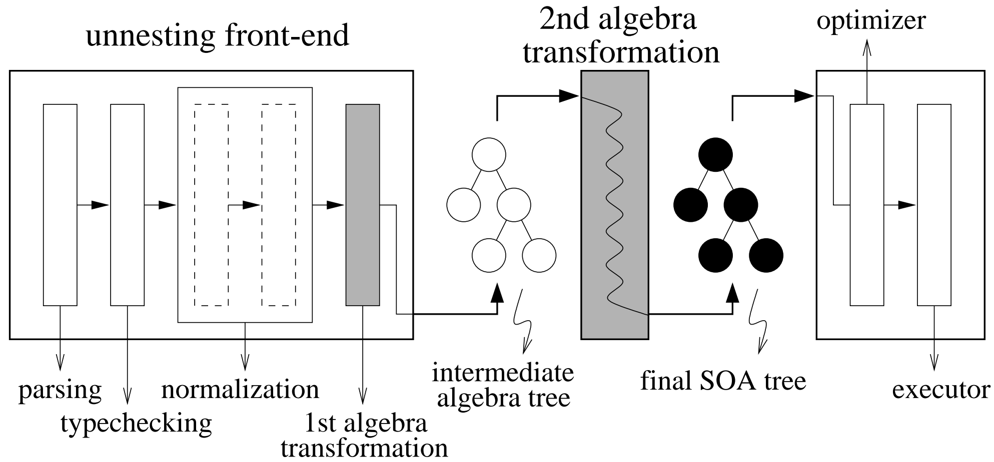

그림 5: OMEGA 시스템의 질의 처리기의 구조

5.1 SOQL 질의 처리기

OMEGA 시스템의 질의 처리기에서는 공간 객체 지원을 위해 OQL을 확장한 공간 객체 질의어인 SOQL(Spatial Object Query Language)을 질의어로 사용하며, 이 SOQL은 공간 객체 대수(SOA, Spatial Object Algebra)[13]를 기반으로 작성되었다.

그림 5는 OMEGA의 질의 처리기에서의 질의 처리 과정을 보여주고 있다. 중첩 구조 제거용 전위 모듈의 1차 대수 변환 과정을 거치면서 중간 논리 대수식을 얻게 되며, 최적화기로 넘겨지기 전에 2차 대수 변환 과정을 거치면서 최적화기에서 요구하는 SOA 트리(SOA tree)로 변환된다. 다음은 SOA 트리를 구성하는 논리 대수 연산자들의 일부를 열거하고 있다. SOQL 질의 처리기의 최적화 모듈에서는 SOA 대수 연산자들만을 인식하며, 1차 대수 변환 과정에서 사용되는 중간 논리 대수 연산자들의 존재에 대해서는 알지 못한다.

- **GET**: 한개의 입력을 가지며, 이 입력에 대한 스키마(schema) 정보를 관리하고 있다. 입력 익스텐트(input extent)에 속하는 모든 객체들을 질의 영역으로 읽어 들인다.

- **SELECT**: 한개의 입력을 가지며, 선택 조건식(selection predicate)을 가지고 있다. 콜렉션에 대해 조건식을 만족하는 객체들만을 선택하여 그 결과를 백(bag) 타입으로 반환한다. 관계 대수의 '`select`' 연산자와 비슷하지만, 입력으로 집합(set)뿐만 아니라, 백(bag), 리스트(list), 배열(array) 타입이 모두 가능하다.

- **JOIN**: 두개의 입력을 가지며, 조인 조건식(join predicate)을 가지고 있다. 두개의 입력에 해당하는 콜렉션으로부터 조인 조건식(join predicate)을 만족하는 객체들의 속성(attribute)을 모두 포함하는 새로운 객체를 생성하고 이들 객체의 백(bag)을 반환한다.

- **MAT**: 경로식(path expression)에서 참조되는 모든 객체들을 질의 영역으로 읽어 들인다. 각 경로(path)에 걸쳐 있는 모든 속성들의 정보를 관리하고 있다.

- **PROJECT**: 한개의 입력을 가지며, 프로젝트(project)될 속성들의 리스트를 관리한다. OQL이나 SOQL의 select절에서는 임의의 수식이 올 수 있으므로, 속성 리스트에 임의의 수식을 표현할 수 있도록 설계되었다.

- **NEST**: 한개의 입력을 가지며, 입력의 평면 속성(flat attribute) a를 중첩(nest)시켜 콜렉션 타입의 속성 a'를 가지는 콜렉션으로 변환한다. 중첩 관계형 데이타 모델(nested relational data model)[14]의 '`nest`' 연산자와 동일하다.

- **UNNEST**: **NEST** 연산자의 역에 해당하며, 입력의 콜렉션 타입의 속성 a를 평면 속성(flat attribute) a'를 가지는 콜렉션으로 변환한다. 이것 역시, 중첩 관계형 데이타 모델의 '`unnest`' 연산자와 동일하다.

질의 최적화기는 2차 대수 변환의 결과인 SOA 트리를 입력으로 받아 최적의 물리 대수식을 만들어 실행기로 넘겨준다. OMEGA 시스템의 질의 최적화기는 Volcano 최적화 생성기(optimizer generator)[15]를 사용한다.

5.2 SOQL 질의 처리기에서의 대수 변환

SOQL 질의 처리기에서의 1차 논리 대수 변환 과정은 4.2절에서 설명한 것과 동일하며, 4.2절에서 열거한 중간 논리 대수 연산자들로 구성된 중간 대수 트리(intermediate algebra tree)를 만들어 낸다. 2차 논리 대수 변환 과정에서는 이 중간 대수 트리의 중간 대수 연산자들을 SOA 연산자들로 치환한 SOA 대수 트리를 생성한다. 중간 논리 대수 연산자들은 모두 조건식을 내포하고 있기 때문에 SOA 연산자로 변환할 때는 반드시 **SELECT** SOA 연산자를 동반하게 된다. 예를 들어, 중간 논리 대수 연산자의 get 연산자는 SOA 연산자인 **SELECT**와 **GET**들로 변환된다. 다음은 중간 논리 대수식을 SOA 대수 트리로 변환하는 과정을 개략적으로 설명하고 있다.

- **get** → **GET-SELECT**
- **join** → **SELECT-JOIN**
- **reduce** → **SELECT-PROJECT**
- **nest** → **SELECT-NEST**
- **unnest** → **SELECT-UNNEST**
- 경로식(path expression) → **MAT**

그림 6은 SOQL 질의를 예로 들어 정규화 과정을 거친 결과를 보여 주고 있으며, 그 이후의 논리 대수 변환 과정은 그림 7에 설명되어 있다.

select struct($E : e$,

M : (select c

from c in $e.children$

where $c.age > 18$))

from e in $Employees$

where $e.name =$ "$KilDong$"

$$\longrightarrow bag\{\ \langle E : e, M : bag\{e \mid c \leftarrow e.children, c.age > 18\}\rangle \mid e \leftarrow Employees, e.name = \text{“}KilDong\text{”}\ \}$$

그림 6: SOQL 예제와 정규화된 모노이드 컴프리헨션

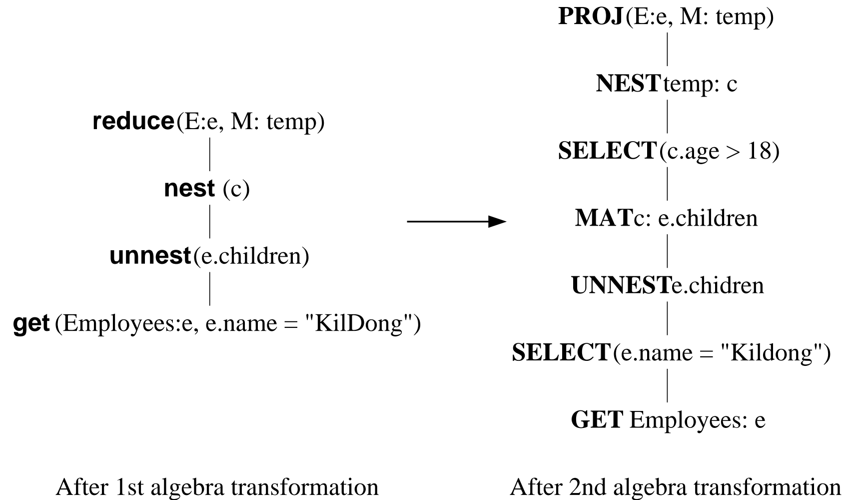

그림 7: 그림 6 예제의 1차 및 2차 논리 대수 변환 결과

6 결론 및 향후 연구 계획

OQL을 비롯한 최근의 객체 질의어들은 대부분 중첩 질의를 지원하고 있으며, SQL과 달리 질의내 위치에 상관없이 자유롭게 중첩 구조를 허용하고 있다. 더군다나, 중첩 질의는 질의 처리기에 상당한 부하를 가져올 수 있는 만큼, 해당 질의 처리기는 질의에 포함된 중첩 구조를 제거하고, 이를 통해 중첩 질의를 보다 효율적으로 처리할 수 있도록 나름대로의 방안을 가지고 있어야 한다. 뿐만 아니라, 현재 중첩 구조를 제거하지 못하는 질의 처리기의 경우 기존의 중첩 객체 대수를 사용하여 중첩 질의를 제거하고자 한다면, 기존 질의 처리기의 최적화 모듈을 전면적으로 수정해야 하는 부담을 져야 한다. 본 논문의 중첩 구조 제거용 모듈은 새로 질의 처리기를 설계 또는 구현하고자 할 때 중첩 구조 제거 관련 모듈을 제공함으로써 구현자로 하여금 부담을 덜어주고, 질의 최적화기와 질의 실행 모듈의 구현에 노력을 집중할 수 있도록 하며, 이미 구현되어 있는 기존 질의 처리기를 확장하고자 할 경우에도 최적화기와 그 이전의 질의 처리 과정에 관여되어 있는 모듈과의 의존성을 최소화 함으로써 이미 구현되어 있는 최적화 모듈의 심각한 수정없이 질의 처리기를 확장할 수 있다는 장점을 제공한다.

현재 이 중첩 구조 제거용 전위 모듈을 사용하여 SOP[5] 시스템의 질의 처리기인 Sopoqles[6]를 확장하고 있으며, 앞으로 중첩 구조 제거용 전위 모듈의 실용성을 높이기 위해 중첩 구조 제거용 전위 모듈 생성기(unnesting front-end generator)를 개발하여 보다 많은 질의 처리 시스템에서 사용할 수 있도록 계획 중이다.

참고문헌

[1] R. G. G. Cattell and D. K. Barry, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, San Francisco, 1997.

[2] Won Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems*, 7(3):443–469, September 1982.

[3] W. Kiessling. SQL-Like and Quel-like Correlation Queries with Aggregates Revisited. Technical Report TR-8, University of California (Berkeley CA), Elec.Research.Laboratory,, September 1984.

[4] R.A. Ganski and H.K.T. Wong. Optimization of Nested SQL Queries Revisited. In *ACM SIGMOD Conf. on the Management of Data 87.*, May 1987.

[5] SNU OODBMS Platform, 서울대학교 컴퓨터공학과에서 개발한 객체지향 데이타베이스 시스템

[6] SOP OQL Extensible System, SOP의 OQL 질의 처리기

[5] U. Dayal. Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates, and Quantifiers. In P. Hammersley, editor, *vldb*, pages 197–208, Brighton, England, September 1987.

[6] M. Muralikrishna. Optimization and Dataflow Algorithms for Nested Tree Queries. In Peter M. G. Apers and Gio Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 77–85, Amsterdam, The Netherlands, 22–25 August 1989. Morgan Kaufmann.

[7] M. Muralikrishna. Improved Unnesting Algorithms for Join Aggregate SQL Queries. In *Proceedings of the 18th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Vancouver*, August 1992.

[8] Sophie Cluet and Guido Moerkotte. Nested Queries in Object Bases. In *dbpl*, September 1993.

[9] Hennie J. Steenhagen, Peter M. G. Apers, Henk M. Blanken, and Rolf A. de By. From Nested-Loop to Join Queries in OODB. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 618–629, Santiago, Chile, 1994.

[10] L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. In *ACM SIGMOD International Conference on Management of Data, San Jose, California*, May 1995.

[11] Leonidas Fegaras. Query Unnesting in Object-Oriented Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27,2 of *ACM SIGMOD Record*, pages 49–60, New York, June1–4 1998. ACM Press.

[12] L. Fegaras. An Experimental Optimizer for OQL. Technical Report TR-CSE-97-007, Department of Computer Science and Engineering, University of Texas at Arlington, 1997.

[13] 박호현, 홍남희, 박창원, and 정진완. OMEGA에서의 공간 객체 대수 및 질의 언어. Technical Report CS/TR-97-118, Department of Computer Science, KAIST, 12월 1997년.

[14] Jan Paredaens and Dirk van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Trans. on Database Systems*, March 1992.

[15] G. Graefe and W.J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proc. of IEEE ICDE*, 1993.

부록

A OQL 2.0의 모노이드 컴프리핸션 해석으로의 변환

다음은 OQL 질의를 모노이드 컴프리핸션 해석으로 어떻게 변환되는지를 OQL을 구성하는 각 문법 구조별로 나타내고 있다.

A.1 OQL 2.0의 변환

A.1.1 생성 수식(construction)

$$\texttt{struct}(l_1 : E_1, ..., l_n : E_n) \longrightarrow \langle l_1 = E_1, ..., l_n = E_n \rangle$$

$$\texttt{set}(E_1, ..., E_n) \longrightarrow \{\ E_1, ..., E_n\ \}$$

$$\texttt{bag}(E_1, ..., E_n) \longrightarrow \{\{\ E_1, ..., E_n\ \}\}$$

A.1.2 한정 수식(quantification), 소속 검사(membership testing)

$$\texttt{for all}\ x\ \texttt{in}\ E_1 : E_2 \longrightarrow all\{\ E_2 \mid x\ \leftarrow\ E_1\ \} \tag{1}$$

$$\texttt{exists}\ x\ \texttt{in}\ E_1 : E_2 \longrightarrow some\{\ E_2 \mid x\ \leftarrow\ E_1\ \} \tag{2}$$

$$E_1\ \theta\texttt{all}\ E_2 (\equiv \texttt{for all}\ x\ \texttt{in}\ E_2 : E_1\ \theta\ x) \longrightarrow all\{\ E_1\ \theta'\ x \mid x \leftarrow E_2\ \} \tag{3}$$

$$E_1\ \theta\texttt{any}\ E_2 (\equiv \texttt{exists}\ x\ \texttt{in}\ E_2 : E_1\ \theta\ x) \longrightarrow some\{\ E_1\ \theta'\ x \mid x \leftarrow E_2\ \} \tag{4}$$

$$E_1\ \texttt{in}\ E_2 \longrightarrow some\{\ E_1 = x \mid x \leftarrow E_2\ \} \tag{5}$$

여기서 $(\theta, \theta') \in \{(=,=), (!=, \neq), (<,<), (<=, \leq), (>=, \geq), (>,>)\}$이고, 위의 규칙 2와 규칙 4에서처럼 존재 한정 수식(existential quantification)은 *some* 모노이드로 변환되는데, 이 존재 한정 수식이 select-from-where 질의에서 나타나면, 그림 1의 규칙 6에 의해 정규화를 거치면서 제거된다.

A.1.3 타입 변환(type conversion)

$$\texttt{flatten}(E) \longrightarrow \mathcal{M}\{\ x \mid e \leftarrow E, x \leftarrow e\ \}$$

$$\texttt{listtoset}(E) \longrightarrow set\{\ x \mid x \leftarrow E\ \}$$

$$\texttt{distinct}(E) \longrightarrow \mathcal{N}\{\ x \mid x \leftarrow E\ \}$$

위에서 모노이드 $\mathcal{M}$과 $\mathcal{N}$은 아래 표와 같이 정해진다.

		e		
		set	*bag*	*list*
E	*set*	set	set	set
	bag	bag	bag	bag
	list	set	bag	list

		distinct(E)
E	*set*	set
	bag	set
	list	list
	array	array

A.1.4 Select-From-Where 질의

select-from-where절로 구성된 질의를 모노이드 컴프리헨션으로 바꿀 때는 다음과 같이 변환한다.

select E

from x_1 in $E_1, ..., x_n$ in E_n

where E_{pred}

$\longrightarrow bag\{\ E \mid x_1 \leftarrow E_1, ..., x_n \leftarrow E_n, E_{pred}\ \}$

select distinct E

from x_1 in $E_1, ..., x_n$ in E_n

where E_{pred}

$\longrightarrow set\{\ E \mid x_1 \leftarrow E_1, ..., x_n \leftarrow E_n, E_{pred}\ \}$

A.1.5 이항 집합 수식(binary collection expression)

E_1 union $E_2 \longrightarrow merge^{\mathcal{M}}(E_1, E_2)$

E_1 intersect $E_2 \longrightarrow \{\ x \mid x \leftarrow E_1, x \in E_2\ \}$

E_1 except $E_2 \longrightarrow \{\ x \mid x \leftarrow E_1, all\{x \neq y \mid y \leftarrow E_2\ \}\ \}$

		E_2	
		set	*bag*
E_2	*set*	set	bag
	bag	bag	bag

여기서, 합집합의 연산 결과의 모노이드 $\mathcal{M}$은 오른쪽 표와 같이 결정된다.