17-2-T01

고차함수 프로그래밍의 원리와 보편성

(Principles and Ubiquity of Programming with Higher-order Functions)

변 석 우
경성대학교 컴퓨터과학과
swbyun@ks.ac.kr

요 약

고차함수는 프로그램의 모듈화 및 합성에 중요한 기능을 하는 함수형 프로그래밍의 대표적 특성 중에 하나이다. 그러나 고차함수 기능처럼 함수를 다른 함수로 전달하는 프로그래밍 기법은 함수형 프로그래밍뿐만 아니라 명령형 프로그래밍에서도 유용한 기능이다. 본 연구에서는 고차함수 프로그래밍의 원리와 실용성에 대해서 살펴본 다음, C 언어의 포인터 기능을 이용하는 경우 고차함수와 유사한 형태의 프로그래밍이 명령형 프로그래밍에서도 가능함을 보인다.

1. 서 론

프로그래밍언어 학자들은 함수형 언어의 이론 및 기술 개발에 많은 연구를 해 왔으며, 그 결과 자동 메모리 관리 (automatic memory management) 기능, 타입 시스템, 고차함수 등의 여러 우수한 기술들을 개발하였다. 함수형 언어 그 자체는 아직도 실용적으로 널리 사용되지 못하고 있지만, 함수형 언어의 기술들은 다른 언어에 적용되어 실용적으로 사용되고 있다. Java가 그 좋은 예이다. Java는 자동 메모리 관리 및 강력 타이핑 (strong typing) 등 함수형 언어에서 연구되었던 기술들을 채택하고 있으며, 많은 Java 프로그래밍의 경험에 비추어 볼 때 이런 기술들의 실용적 중요성은 이미 검증되었다고 볼 수 있다. 향후 우수한 함수형 언어의 핵심 기술인 HOT (Higher-order function and Types) 기술은 다양한 방법으로 활용될 수 있을 것으로 전망한다 [1].

일반적으로 프로그래밍 언어 기술은 반드시 어떤 특정 언어에서만 이용될 수 있도록 종속적인 것은 아니다. 예를 들어, 타입이나 고차함수 등의 기술은 함수형 언어를 중심으로 개발되었으나, 다른 언어에서도 이 기술들을 이용할 수 있을 것이다.

본 연구에서는 고차함수 프로그래밍의 원리와 보편성에 대해서 논의한다. 고차함수는 함수를 자유롭고 유연하게 사용할 수 있도록 함으로써 프로그램의 모듈 및 합성에 유용한 역할을 한다. 고차함수는 함수형 언어에서 채택되었지만 다른 언어에서도 채택해도 좋을 만큼 우수한 기술이라고 보인다. 본 연구에서는 C의 포인터

를 이용함으로써 고차함수 프로그래밍과 유사한 효과를 가질 수 있음을 보인다, 또한, 고차함수 프로그래밍의 의미가 함수를 자유롭고 유연하게 이용할 수 있도록 하는 데 있다고 볼 때, 객체지향 프로그래밍 또한 같은 맥락에 있다고 볼 수 있다.

본 연구에서는 2절에서 고차함수 프로그래밍을 가능케 하는 이론 배경 및 원리에 대해서 설명한 다음, 3절에서는 고차함수 프로그래밍의 중요성에 대해서 논의한다. 4절에서는 C의 포인터를 이용하여 고차함수 프로그래밍과 유사한 효과를 가질 수 있는 프로그램 예를 소개하며, 마지막으로 향후 연구 동향과 결론을 제시한다.

본 연구에서는 독자들이 C와 함수형 프로그래밍에 대한 기본적인 경험과 지식을 갖고 있음을 전제로 논의를 전개한다.

2. 고차함수 프로그래밍의 원리

2.1. 고차함수

함수는 프로그래밍 추상화의 가장 기본적인 기능으로서 함수형 프로그래밍뿐만 아니라 명령형 프로그래밍에서도 함수를 표현할 수 있는 환경을 제공하고 있다. 명령형 프로그래밍의 프로시저는 함수의 기능을 표현하기 위한 구문이다. 그러나 명령형 언어에서는 Haskell과 같은 순수 함수형 언어와는 달리 전역 변수 등을 사용하는 경우 한 식 (expression)의 값이 유일하지 않고 문맥 (context)에 따라 변하는 문제가 발생한다. 또한, 함수형 프로그래밍에서의 함수 표현 방식은 명령형 프로그래밍의 경우보다 훨씬 자유롭다. 명령형 프로그래밍과는 달리 함수형 프로그래밍에서 함수가 일등급 시민 (first-class citizen) 으로서 표현될 수 있다. 마치 수나 문자처럼, 함수가 다른 함수의 인수로서 나타날 수 있으며, 저장될 수 있고, 복귀 값으로 이용될 수도 있다. 이러한 형태의 함수를 고차함수 (higher-order function) 라고 부른다.

고차함수의 표현 방법과 계산 원리는 현재 수학에서 보통 사용하는 방식과는 다르다. 고차함수의 원리는 1930년대 연구되었던 람다 계산법에서 찾을 수 있으며, SML과 Haskell과 같은 언어는 람다 계산법의 원리에 따라 충실히 설계되었다고 할 수 있다.

그러나 SML과 Haskell을 람다 계산법과 직접적으로 연관시키는 데는 몇 가지 기술적인 문제가 있다. 무엇보다도 함수형 언어는 람다 계산법에서 사용하지 않는 함수 기호를 사용하고 있으며, 수학적 특징의 가장 대표적인 기능인 등식 이론 (equational theory) 원리를 직접적으로 적용하기 어려운 문제가 있다. 본 연구에서는 람다 계산법 대신 TRS (Term Rewriting Systems, 항 개서 시스템)을 이용하여 고차함수의 원리를 설명한다.

2.2. Applicative Term Rewriting Systems

TRS에 대한 자세한 소개는 [2]와 [3]을 참조하기 바란다. TRS는 기호들의 집합인 $\sum$와 재작성 룰인 R의 쌍 ($\sum$, R)으로서 구성된다. 기호들은 인수의 수가 정해진 함수 기호의 집합으로서 구성된다. $\sum$가 주어지면 그에 대한 항 (term)은 함수기호와 변수로서 구성된다. 각 변수는 항이 될 수 있으며, F가 n개의 인수를 갖는 함수기호이고 t_1, ..., t_n 이 항일 때 $F(t_1, ..., t_n)$ 또한 항이다. Ter($\sum$)는 이와 같은 항들의 집합을 의미한다. 재작성 룰은 두 개의 항 s와 t가 주어졌을 때 이들을

$$s \rightarrow t$$

형태로 표현함으로써 구성된다. 이때 룰의 왼쪽에 있는 항 s는 변수가 될 수 없다. R은 이러한 룰들의 집합이다.

한 항에 같은 변수의 이름이 단 한번만 나타

나는 경우 그 항은 선형(linear)이라고 부르며, 룰의 왼쪽 항이 선형인 경우 그 룰은 좌선형(left-linear)라 부른다. R의 각각의 룰들이 좌선형일 때 R은 좌선형이라고 부른다.

여기서 → 관계는 반사적(reflexive)이며 추이적(transitive)인 성질을 갖는데, 직관적으로 이야기하면 등식 기호 (=)에서 대칭형(symmetric) 특성을 제거함으로써 방향성을 갖도록 한 것이다. 즉, 이것은 방향성을 갖는 등식 관계로서 *축약 (reduction)*이라고 불린다. 축약은 함수 이론에 대한 기본적인 계산 원리로서 람다 계산법에서도 이용되고 있다.

TRS에서 항은 함수형 (functional) 방식과 적용형(applicative) 방식으로 표현될 수 있다. 함수형이란 수학에서 일반적으로 이용하고 있는 대로 각 함수 기호의 인수의 수 (arity)를 존중하여 표현하는 방법이다. 즉, 각 함수 기호마다 미리 정해진 인수의 수가 있으며, 항을 구성할 때는 정해진 인수의 수를 정확히 제공해야 한다. 예를 들어, 이진 함수 *plus*에 대해서는 *plus(x, y)*, *plus(plus(1, x), y)* 등은 합법적인 형태를 취하고 있으나, *plus(x)*와 같은 형태로 표현될 수는 없다. 현대 수학의 대부분은 함수형 TRS의 형태로서 표현되고 있으며, TRS 또한 함수형 형태가 주류를 이룬다.

적용형(applicative) TRS (ATRS)는 1958년 Curry와 Feys에 의해서 소개되었다 [4]. ATRS의 항은 함수형 TRS와는 달리 단 한 개의 이진 함수인 *Ap*를 제외한 모든 함수의 인수의 수를 0으로 취급한다. 예를 들어, *plus(x, y)*에 해당하는 ATRS의 텀은 *Ap(Ap(plus, x), y)*으로서, *plus*의 인수는 0이다. 정형적으로 ATRS는 TRS의 일종으로서 다음과 같이 정의된다.

[ATRS 정의]

(1) 어떤 기호들의 집합 ∑가 주어졌을 때, 그에 대한 적용 항 (applicative term)은 ∑의 모든 기호들 각각의 인수의 수 (arity)를 0으로 만들고 하나의 이진 함수 기호 *Ap*를 첨가한 기호들로서 구성되는 항으로 만들어 진다. 이때 *Ap*는 ∑에는 속하지 않는 특수한 기호이다. ATer(∑)은 ∑에 대한 적용 항들의 집합을 의미한다.

(2) 함수 $cur : Ter(\Sigma) \rightarrow ATer(\Sigma)$은 재귀적으로 다음과 같이 정의된다:

$cur(x) = x$

$cur(F(t_1,\cdots,t_n)) = Ap(\cdots Ap(F, t_1),\cdots, t_n)$

(3) ATRS는 두 적용 항들로서 구성된 룰들의 TRS이다.

ATRS의 대표적인 예로서 CL (Combinatory Logic)이 있다. CL은 세 개의 인수를 갖는 오퍼레이터 *S*, 두개의 인수를 갖는 *K*, 그리고 한 개의 인수를 갖는 *I*와 다음과 같은 룰로서 구성된다.

$Ap(Ap(Ap(S, x), y), z) \rightarrow Ap(Ap(x,z), Ap(y, z))$

$Ap(Ap(K, x), y) \rightarrow x$

$Ap(I, x) \rightarrow x$

ATRS의 표현은 다음과 같은 표현상의 규칙을 적용하여 간단하게 표시할 수 있다.

(1) $Ap(t, s)$는 Ap 대신 · 기호를 중간에 위치하여 $t \cdot s$로서 표현한다.

(2) 대수학에서 일반적으로 사용하는 방식대로 · 표현을 생략한다.

(3) 괄호는 왼쪽부터 (left-associative) 적용한다.

그러면 위의 CL은 다음과 같이 표현된다.

$Sxyz \rightarrow xz(yz)$

$Kxy \rightarrow x$

$Ix \rightarrow x$

이때 *Sx, Sxy, Sxyz* 등이 모두 구문적으로 합법적이므로 마치 오퍼레이터들이 변형된 수의 인수를 가질 수 있는 것처럼 생각할 수 있다.

ATRS에서 정의된 함수 *cur*은 함수형 TRS의 항을 ATRS 항으로 변환시키는 일을 한다. 이 과정에서 우리는 어떤 TRS 룰 R에 대해서 *cur(R)*이 본래 R의 특성을 제대로 반영하는 지에 대한 의문을 가질 수 있다. 즉, R에서 계산할 때 나타나는 여러 현상들이 *cur(R)*에서도 같은 모습으로 나타나는지의 문제이다. 만약 같지 않다면 R을 커리화(currying)로 변환한 *cur(R)*을 서로 연관시켜 논의할 수는 없는 것이다. 이것은 곧 어떤 문제에 대해서 그것을 ATRS 형태인 함수형 언어로 프로그래밍하는 경우 "함수형 프로그래밍이 원래의 문제를 정확하게 계산해 낼 수 있는가?"에 의미를 갖는다. 즉, 함수형 프로그래밍의 안전성 문제와 연결된다.

[5]에 따르면, 좌선형 (left linear) TRS인 경우 R과 *cur(R)* 사이의 계산 특성은 일치하지만 비좌선형 (non-left linear) TRS에서는 불일치의 문제가 발생할 있음을 증명하였다. 함수형 프로그래밍은 좌선형의 특성을 지니므로, ATRS에 의한 함수형 프로그래밍의 안전성 문제는 발생하지 않는 셈이다.

2.3. ATRS로서의 함수형 언어

SML이나 Haskell과 같은 함수형 언어들의 항과 룰들은 ATRS의 원리를 따른다. ATRS를 이용함으로써 괄호의 수를 줄이고 간략한 표현을 할 수 있으며, 또한 함수에 대한 인수의 수를 고정된 형태가 아닌 변형된 형태로 표현할 수 있도록 한다. *plus, plus 1, plus 1 2* 가 모두 구문적으로 합법적이다. 그러므로 이들은 다른 함수의 인수가 될 수 있고, 복귀 값의 형태로 이용될 수도 있으며, 리스트 등에 저장될 수도 있다.

이와 같이, 함수형 언어의 고차함수의 원리는 ATRS로서 자연스럽게 설명될 수 있다. 그러나, 함수형 언어의 모든 것이 ATRS와 직접적으로 연관되는 것은 아니다. 예를 들어, Miranda™ 에서는 다음과 같은 룰들을 정의할 수 있다.

```
eq x x = true
loop = loop
```

이 룰에서 loop은 계산이 영원히 종료되지 않는 룰을 의미하며, 왼쪽 항에 변수가 두 번 나타났으므로 비좌선형인처럼 보인다. 앞서 언급한 대로 비좌선형인 경우 TRS를 커리화한 ATRS는 원래의 TRS의 계산을 제대로 표현하지 못할 수 있다. 그러나, TRS와 Miranda™ 에서의 비좌선형의 의미는 일치하지 않는다. TRS에서 같은 변수가 두 번 이상 반복하여 나타나는 것은 두 식의 문자들이 일치함을 의미하지만, Miranda™ 에서는 두 식의 '값'이 같음을 의미한다. 즉, 위의 룰에서, (eq loop loop)라는 식이 주어졌을 때 ATRS에서는 이 결과 값을 true로 계산하지만, Miranda™ 에서는 eq의 두 인수 (loop)의 값을 먼저 계산하려고 함으로써 영원히 그 값을 계산하지 못하게 된다. 결과적으로, Miranda™ 와 같은 형태의 함수형 언어는 구문적으로 비선형 형태의 룰을 사용하고 있으나, ATRS나 TRS의 비선형의 의미와는 다르므로 선형이라고 보는 것이 타당하다.

ATRS가 좌선형이 아닌 경우 Church-Rosser 특성이 성립하지 않는 등의 중대한 문제가 발생할 수 있다. Miranda™ 와 같은 일부 함수형 언어에서 구문적으로 비좌선형이 형태의 룰을 정의할 수 있지만 TRS 적인 의미의 비선형은 아니므로 Church-Rosser의 특성은 유지된다. 최근에 개발된 Haskell과 같은 함수형 언어에서는 룰의 왼쪽에 같은 변수의 이름이 여러 번 사용되는 것을 제한하고 있다.

함수형 언어가 ATRS와 다른 또 하나의 문제

로서 룰의 패턴 매칭이 선언적이지 못한 문제가 있다. 예를 들어, 팩토리얼을 계산하는 함수 fac은 Haskell로 다음과 같이 표현될 수 있다.

```
fac 0 = 1
fac n = n * fac (n-1)
```

여기서 = 은 표현 기호만 다를 뿐 ATRS의 →와 같은 의미로 사용된다. 문제가 되는 것은 ATRS의 룰에 대한 패턴 매칭이 룰의 위치와 관계없이 진행되는 완전한 선언적 (declarative) 특성을 갖는 것에 비하여 함수형 프로그래밍에서는 룰에 대한 패턴 매칭이 룰의 위로부터 아래 방향으로 진행된다는 점이다. 위의 fac룰에 대해서 (fac 0)와 같은 항이 주어 졌을 때 ATRS에서는 fac의 첫 번째 룰과 두 번째 룰 모두와 패턴 매칭되지만, 함수형 프로그래밍에서 이 항은 언제나 첫 번째 룰과 패턴 매칭된다.

함수형 프로그래밍의 계산 방식이 TRS (혹은 ATRS)의 원리와 완전히 일치하지 않음에 따라 TRS의 대표적인 장점인 등식 추론의 원리 및 여러 기초 이론을 함수형 언어와 연관시키는 데는 몇 가지 어려운 점이 있다. 자세한 설명을 원하는 독자는 [6]을 참조하기 바란다.

3. 고차함수 프로그래밍의 중요성

John Hughes가 고차함수 기능이 프로그램의 모듈화 및 합성에 중요한 역할을 할 수 있다는 주장이 제기한 이래 [7], 고차함수의 기능은 함수형 프로그래밍의 핵심적 기능으로서 인정되고 있다.

프로그램 모듈의 정의/합성을 위한 고차함수 기능의 유용성을 고차함수의 기능을 사용하지 않는 명령형 언어와 고차함수의 기능을 갖는 Haskell로 표현하는 경우를 비교하여 설명한다. 다음과 같은 명령형 프로그램을 고려해보자.

```
for (i = 0; i < n; i++) {
  a[i] = sqrt (a[i])
}
```

이 프로그램은 for 문을 사용하여 배열 a의 각 요소에 있는 정수에 sqrt 함수를 적용하는 (apply) 프로그램이다. 이와 같은 의미의 프로그램을 Haskell로 표현한다면 map 함수를 이용할 것이다.

```
map f [] = []
map f  (x:xs)  = (f x) : map f xs
```

이렇게 정의된 map 함수를 이용하여,

map sqrt [a_1, ..., a_n]

을 수행함으로써 리스트 a에 있는 각 원소들에게 sqrt 함수를 적용하도록 한다. map은 함수 f와 리스트 형태로 된 데이터들 (x:xs)를 받아들여, 받아들인 함수가 데이터 각 원소에 적용되도록 하는 것으로서, 이 기능은 C 언어의 for문에 해당된다고 불 수 있다. map의 f에 바인딩되는 것은 함수이므로 map은 인수로서 함수를 이용하는 고차 함수의 기능을 이용하고 있다.

map과 for의 차이점으로서, for는 단순히 for 그 자체만으로는 표현될 수 없고 반드시 적용될 함수와 데이터가 함께 표현되어야 하며 - 위의 예에서는 배열 a와 함수 sqrt - 따라서 이러한 패턴의 프로그램이 나올 때마다 for 문을 반복해서 작성해야 한다는 점이다. 이에 비해서, 함수형 프로그래밍의 map은 한번만 정의되면 그것을 다시 정의할 필요 없이 재사용할 수 있으며, 따라서 코드의 중복을 피할 수 있고, 프로그램을 간결하게 표현할 수 있게 한다. 위의 경우, sqrt는 함수의 인수로서 전달되어 사용되는 데, map은 단지 sqrt 뿐만 아니라 모든 unary 함수

를 입력받을 수 있다. 예를 들어, 리스트 b의 각 원소의 값을 3만큼 증가시키려 할 때는 이미 정의된 map을 재사용하여 단순히 map (+3) [b_1, ..., b_n] 으로서 표현하면 된다.

프로그램의 모듈화와 합성은 상호 보완적 관계를 갖는다. 프로그램 모듈화를 극대화하기 위해서는 정교한 합성 기술이 필수적인데, 고차함수 기능은 정교한 함수 (혹은 프로그램) 합성을 가능케 한다. 위의 map 함수의 예에서, 순환 기능을 하는 함수와 각 원소에 적용(application)되는 함수를 따로 분리(모듈화) 함으로써 코드를 재사용할 수 있게 하고 프로그램의 표현을 간결하게 한다. 이와 같이 고차 함수는 프로그램의 모듈화와 합성에 중요한 역할을 한다.

위의 map과 같이, 함수들을 합성하기 위한 목적으로 정의되는 함수들을 컴비네이터(combinator)라 부른다. 컴비네이터는 고차 함수 기능을 이용하여 합성하려는 함수들을 컴비네이터의 인수로서 입력받도록 하는 형태로 정의된다. 널리 사용되는 대표적인 컴비네이터로서 map 외에 fold가 있다. fold에 대한 설명은 함수형 프로그래밍 관련 자료들을 참조하기 바란다 [8]. fold는 매우 막강한 표현력을 가지고 있어, 대부분의 재귀함수가 fold로서 표현될 수 있다 [9]. map과 fold는 대표적인 순환 컴비네이터 (iterator)이다.

고차 함수의 실용적 중요성은 프로그램 모듈의 정의 및 이들의 합성이 모든 프로그램 개발에 적용될 수 있는 보편적인 방법이라는 점에서 찾을 수 있다. 각 응용 분야의 특성에 따라 다양한 종류의 컴비네이터들이 소개되고 있다. 멀티미디어 동영상 프로그래밍 환경을 지원하기 위한 Fran (Functional Reactive ANimation)에서는 over 컴비네이터를 사용하고 있다 [10][11]. over는 동영상으로 출력되는 두 함수를 입력받아 이 둘을 하나의 화면에 합성시켜 출력하도록 하는 기능을 가지고 있다.

또한, 금융 보험 분야의 계약서 작성에 함수형 언어의 컴비네이터 라이브러리가 매우 유용하다는 연구 보고가 발표되었다 [12]. 일반적으로 금융 보험 분야의 계약은 복잡하고 방대하다. 한 금융 보험 계약은 많은 하부 계약들로 구성되고, 하부 계약 역시 더 하위의 계약으로 구성되는 과정이 반복된다. 이 과정은 대형 프로그램을 구성하는 방법과 일치하므로, 복잡한 계약서의 내용을 정확하게 '계산'해내기 위한 컴비네이터 기술을 적용할 수 있다.

4. 포인터를 이용한 고차함수 프로그래밍

명령형 프로그래밍에서는 고차함수의 기능을 지원하지 못하고 있으나, C의 포인터와 같이, 함수에 대한 주소 값을 이용하여 함수의 주소를 다른 함수로 전달하고 저장함으로써 고차함수와 같은 효과를 갖도록 할 수 있다. 이 절에서는 C의 포인터를 이용한 프로그래밍 방법에 대해서 논의한다.

4.1. 함수의 포인터를 인수로 이용

다음 [예제-1]에서 iterator는 함수 포인터와, 배열 및 그 배열의 수에 대한 정보를 인수로 받아 들여 주어진 함수가 배열의 각각의 인수에 적용되도록 하는 기능을 한다. 예로서 이 프로그램에서는 arr[1,2,3]의 배열을 함수 plus2와 prod3를 각각 적용한다.

예제-1에서 iterator (plus2, 3, arr)를 수행한 후 배열 b의 값은 [3,4,5]이며, iterator (prod3, 3, arr)을 수행하면 b의 값은 [3,6,9]가 된다. 이것은 각각 함수형 프로그래밍의 map (+2) [1,2,3]과 map (*3) [1,2,3]을 수행한 결과와 유사함을 알 수 있다. map 대신 foldr 함수를 이용하여 각각 다음과 같이 표현할 수 있다.

foldr (\x xs -> (+2) x : xs) [ ] [1,2,3]
foldr (\x xs -> (*3) x : xs) [ ] [1,2,3]

[예제-1]

```
#include <stdio.h>

int plus2 (int x);
int prod3 (int x);

main() {
 int arr[] = {1, 2, 3};
 iterator (plus2, 3, arr);
 iterator (prod3, 3, arr);
}

iterator (int (*f)(int), int n,
int *a)
{
 int i; int b[3];
 for (i = 0; i < n; i++)
  { b[i] = f(a[i]); }
}

int plus2 (int x)
  {return x + 2;}

int prod3 (int x)
  {return x * 3;}
```

4.2. 함수의 포인터를 복귀 값으로 이용

다음 예제-2에서 test1의 수행결과 값은 test2의 포인터이다. test1을 호출한 결과 값을 g로 받아서 이 포인터의 함수를 호출하면 결과적으로 test2 함수를 호출하는 것과 같은 효과를 갖는다. 따라서 다음 프로그램을 수행 시키면 22라는 글자가 화면에 출력된다.

[예제-2]

```
#include<stdio.h>

void* test1();
void test2();

main() {
  void (*g)(void);
  g = test1();
  g();
}

void* test1() {return test2;}

void test2() {printf("22\n");}
```

4.3. 함수의 포인터를 배열로 저장

[예제-3]

```
#include<stdio.h>

void test();

main(){
       void (*g[2])(void);
       g[1] = test;
       (*g[1])();
}

void test(){ printf("dd\n"); }
```

예제-3은 test 함수의 포인터를 배열 g에 저장한 다음, 나중에 배열에 저장된 함수를 호출하는 프로그램이다. 결과적으로 (*g[1])()은 test()와 같은 효과를 가지며, dd 라는 글자가 화면에 출력된다.

5. 결론 및 향후 연구 방향

본 연구에서는 함수형 프로그래밍과 고차함

수의 기본 원리를 TRS의 측면에서 설명하고, 고차함수와 유사한 효과를 갖는 명령형 프로그래밍 기법을 C의 포인터를 이용하여 소개하였다. 고차함수는 실용적으로 프로그램의 합성/모듈화 및 컴포넌트 프로그래밍에 유용하다. 실제로 본 연구에서 소개한 형태의 프로그래밍 기법은 컴포넌트 프로그래밍의 인터페이스에 사용되고 있다 [13]. 함수형 언어가 프로그램 합성을 위한 스크립팅 언어로서 적합하다는 주장[14]은 기술적으로 타입과 함께 함수형 프로그래밍의 고차함수 기능을 강조하는 것이라고 볼 수 있다. 명령형 프로그래밍 언어도 이런 기능을 갖춤으로써 프로그램의 합성 및 스크립팅 기능이 향상될 수 있다.

람다 계산법 기반으로 함수형 언어를 설계 구현하는 것처럼, 명령형 프로그래밍 언어에 함수형 프로그래밍에서 개발된 기술들을 적용시키기 위해서는 명령형 프로그래밍을 위한 이론적 체계를 갖추고 프로그래밍 특성을 정형화하는 연구가 필요할 것이다.

감사의 글

본 연구는 한국과학재단 목적기초연구 (R01-2000-00287) 지원으로 수행되었습니다.

참고문헌

[1] Philip Wadler. A HOT opportunity. Editorial, *Journal of Functional Programming*, 7(2):127-128, March 1997.

[2] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, vol. B, Chapter 15. North-Holland. 1989.

[3] J. Klop. Term rewriting systems, In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, vol. II, Oxford University Press, 1992.

[4] H. Curry and R. Feys. *Combinatory Logic*. vol. I. North-Holland, 1958.

[5] R. Kennaway, J. Klop, R. Sleep, and F. de Vries. Comparing curried and uncurried rewriting. J. of Symbolic Computation, vol. 11, 1998.

[6] R. Kennaway. The specificity rule for lazy pattern-matching in ambiguous term rewriting systems. In ESOP '90, *Lecture Notes in Computer Science* no. 432, 1990.

[7] John Hughes. Why functional programming mattes. *Computer Journal*, 32(2) : 98-107, 1989.

[8] Richard Bird and Oege de Moor, *Algebra of Programming*. Prentice -Hall, 1997.

[9] 변석우. 고계 함수화 컴비네이터의 프로그래밍 원리. 한국정보과학회 프로그래밍언어 논문지, 제 16권 제 1호, 25-34. 2002년 2월.

[10] http://conal.net/fran/turorial.html

[11] Paul Hudak. *The Haskell School of Expression : Learning Functional Programming Through Multimedia*. Cambridge University Press, 2000.

[12] Simon Peyton Joes, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering. *Proc. of International Conference on Functional Programming*, 2000. (http://haskell.org/practice.html)

[13] Dale Rogerson. *Inside COM*. Microsoft Corporation. 1997.

[14] Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM Components in Haskell, *In Proc. International Conference on Software Reuse*, 1998.

변 석 우

1976~1980. 숭실대학교 전자계산(학사).
1980~1982. 숭실대학교 전자계산(석사).
1982.~1999. ETRI 책임연구원
1988~1994. 영국 University of East Anglia 전산학(박사).
1999.~현재 경성대학교 컴퓨터과학과 부교수

관심분야: rewriting system, 함수형 프로그래밍, 프로그래밍언어 의미론 등.