고계 함수와 컴비네이터의 프로그래밍 원리 (Programming Principles of Higher-Order Functions and Combinators)

변 석 우
경성대학교 컴퓨터과학과
swbyun@ks.ac.kr

요 약

함수형 프로그래밍의 고계 함수 기능은 프로그램의 합성에 탁월한 기능을 하며 프로그램 코드의 재사용을 극대화시킬 수 있도록 한다. 컴비네이터는 함수들을 합성하기 위한 목적으로 정의된 함수로서 고계 함수 기능을 이용하여 정의된다. 프로그래밍의 패턴에 따라 다양한 형태의 프로그램 합성 기법과 컴비네이터들이 정의될 수 있다. 대표적인 컴비네이터로서 map, fold, 모나드 (>>=) 등이 있으며, 새로운 컴비네이터들이 계속 연구되고 있다.

본 연구에서는 컴비네이터의 기본 원리에 대해서 설명하고, 재귀 함수들을 fold를 이용하여 정의하는 원리에 대해서 논의한다. fold를 재사용하여 다양한 재귀 함수를 정의할 수 있다. 논의되는 내용들을 Haskell로 프로그래밍 함으로써 그 원리의 타당성을 검증하였다.

1. 서론

일반적으로 프로그램의 개념을 수동적인 '데이터'와 능동적인 역할을 '함수' (혹은 '프로시듀어')로 구분하는 경향이 있다. 프로그램을 데이터와 이들을 입력받아 프로세싱하는 함수의 형태로 되어 있다고 보는 것이다. 그러나. 함수형 프로그래밍에서는 이러한 이분법적인 개념이 성립하지 않는다. 고계 함수 (higher-order function)를 사용하는 함수형 프로그래밍에서 함수는 함수의 인수 (parameter)와 복귀값 (return)으로서 이용될 수 있으므로, 함수는 능동적이면서 동시에 수동적인 개념이다.

고계 함수 개념의 근본은 람다 계산법 (lambda calculus)에서 찾을 수 있다. 람다 계산법은 함수에 대한 계산 이론 체계로서, 람다 계산법에서는 모든 객체가 함수로 표현될 수 있음을 보이고 있다. 예를 들어, 우리가 흔히 알고 있는 함수뿐만 아니라, 정수 (0, 1, 2, ...), 부울 값 (True, False) 등도 람다 식으로 표현하고 있다[1]. 람다 계산법을

기반으로 설계된 SML, Haskell 등의 언어들은 자연스럽게 고계 함수의 기능을 갖고 있다.

고계 함수 기능의 실용적인 중요성은 프로그램 합성 (synthesis)에서 찾을 수 있다. 일반적으로 프로그램은 프로그램 부분(fragments)들을 합성하는 과정을 반복적으로 수행함으로써 구성된다. 이때 프로그램 부분들은 기능적으로 중복됨이 없이 재사용할 수 있도록 하는 형태로 정의되는 것이 바람직하다. 이것을 위해서는 프로그램 부분들을 세밀하게 분리하여 정의하는 것이 필요한데, 프로그램의 분리 정도는 프로그램의 합성 능력에 달려 있으므로 프로그램 부분들의 정의와 이들을 합성하는 것은 밀접한 상관관계를 갖고 있다. 함수형 프로그램의 고계 함수 기능은 고도의 프로그램 합성을 가능케 하며, 그 덕택으로 함수형 프로그래밍에서는 코드의 중복을 방지할 수 있고 프로그램을 간결하게 표현할 수 있다[2].

예를 들어, 다음과 같은 C 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 프로그램이다. 이것을 유사한 의미의 Haskell 함수로 코딩한다면 map 함수를 이용할 것이다.

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

이렇게 정의된 map 함수를 이용하여,

```
  map sqrt [a1, ..., an]
```

을 수행함으로써 리스트 a에 있는 각 원소들에게 sqrt 함수를 적용하도록(apply) 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 볼 수 있다. map의 f에 바인딩되는 것은 함수이므로 map은 고계 함수 기능을 이용하여 정의되고 있음을 알 수 있다. map과 for의 차이점으로서, for는 단순히 for 그 자체만으로는 표현될 수 없고 반드시 적용될 함수와 데이터가 함께 표현되어야 하며 - 위의 예에서는 a와 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 재 사용할 수 있으며, 따라서 코드의 중복을 피할 수 있고, 프로그램을 간결하게 표현할 수 있게 한다. 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. 이와 같이, 고계 함수 기능이 코드의 재사용과 간결성에 중요한 역할을 한다는 사실을 알 수 있다.

위의 map과 같이, 함수들을 합성하기 위한 목적으로 정의되는 함수들을 컴비네이터 (combinator)라 부른다. 컴비네이터는 고계 함수 기능을 이용하여 합성하려는 함수들을 컴비네이터의 인수로서 입력받도록 하는 형태로 정의되는데, 합성의 패턴에 따라 다양한 형태의 컴비네이터들이 정의될 수 있다. 대표적인 컴비네이터로서 map, fold, 모나드 (>>=) 등이 있으며, 다양한 응용 프로그래밍이 개발됨에 따라 새로운 컴비네이터들이 계속 연구되고 있다.

본 연구에서는 고계 함수의 기본 원리에 대해서 설명하고, 재귀 함수들을 fold를 이용하여 정의하는 원리에 대해서 논의한다. 자연수 및 리스트에 대해서 정의되는 재귀

함수들은 fold를 이용하여 정의될 수 있다. 또한 고계 함수의 기본 원리를 람다 계산법과 적용 항 개서 시스템 (applicative term rewriting system)과 연관시켜 설명한다.

마지막으로, 모나드 컴비네이터를 이용하는 프로그래밍 합성의 기본 개념과, 함수를 Haskell이 아닌 C나 Java 등의 타 언어 구현하는 경우 이들의 합성에 대해서 간략히 논의한다.

2. Currying

2.1. 람다 계산법에서의 고계 함수

오늘날 수학에서 함수에 대해서 적용되고 있는 원칙 중의 하나로서, 함수는 고정된 인수의 수 (arity)를 갖도록 정의되고 있다. 예를 들어, 더하기 함수 plus는 반드시 두 개의 인수를 갖도록 plus(x, y) 형태로 정의된다. 그러나, 1930년대 람다 계산법에서는 이와는 다른 개념으로 함수를 정의하였다.

람다 계산법에서 함수의 표현은 고정된 인수의 수를 가질 필요 없이 자유롭게 표현된다. 예를 들어, plus, (plus 1), (plus 1 2), (plus 1 2 3), (plus 1 2 3 4 ..... 100), ... 등이 모두 '정당한' 람다 계산법의 구문인 것이다. 현재 이런 개념의 함수는 수학의 주류는 아니지만 프로그래밍 언어의 중요한 이론적 기반으로서 인식되고 있다. 람다 계산법에서 모든 객체는 함수로서 자연수나 부울 값 역시 함수이다.

예를 들어, 처치의 수 (Church numeral)에서 정수 n (n≥0)은 다음과 같이 표현된다.

$n = \lambda f\ x.\ f\ (f\ (f\ \ldots\ (f\ x)\ \ldots\ ))$

여기서 f는 n번 나타나는데, 이를 간략하게

$n = \lambda f\ x.\ f^n\ x$

으로 표현한다. 모든 사칙 연산자 또한 람다 계산법으로 표현될 수 있는데, 처치의 수로 표현되는 정수에 대해서 plus는 다음과 같이 표현된다.

$plus = \lambda mnab.\ (m\ a)\ (n\ a\ b)$

이 정의의 타당성은 다음과 같은 예를 통해서 쉽게 점검할 수 있다.

$plus\ 2\ 3$
$\equiv (\lambda mnab.\ (m\ a)\ (n\ a\ b))\ (\lambda fx.\ f\ (f\ x))\ (\lambda fx.\ f\ (f\ (f\ x)))$
$= \lambda ab.\ ((\lambda fx.\ f\ (f\ x))\ a)\ ((\lambda fx.\ f\ (f\ (f\ x)))\ a\ b)$
$= \lambda ab.\ (\lambda x.\ a\ (a\ x))\ (a\ (a\ (a\ b)))$
$= \lambda ab.\ a\ (a\ (a\ (a\ (a\ b))))$
$\equiv \lambda ab.\ a^5\ b\ (\equiv 5)$
(여기서 ≡은 두 항이 동일함을 의미한다).

어떤 한 람다 식 M이 람다 식 N에 적용된 형태는 (MN)으로 표현하는데, 프로그래밍의 관점에서 볼 때 M은 함수이고 N은 실인수 (actual parameter)로서 볼 수 있다. 위의 plus 2 3의 계산 과정에서, 수 2 와 3에 해당하는 람다 식이 다른 람다 식에 적용되어 $((\lambda fx.\ f\ (f\ x))\ a)$와 $((\lambda fx.\ f\ (f\ (f\ x)))\ a\ b)$ 같은 람다 식이 나타나고 있음을 알 수 있다. 이와 같이, 자연수 또한 일반 함수와 똑 같은 형태로 동작하고 있다.

2.2. 적용 항 개서 시스템

람다 계산법의 구문(syntax)은 변수, λ-

추상화 (λ-abstraction), 적용 (application)의 지극히 간단한 형태로 구성된다. 이러한 간단한 구문적 체계에도 불구하고 함수에 대한 대부분의 개념을 제대로 표현하고 있는 점이 람다 계산법의 장점이며 성공요인으로서 인정되고 있다.

흔히 SML이나 Haskell과 같은 함수형 언어를 람다 계산법의 '구문적 치장(syntactic sugaring)'이라고 말하고 있다. 이 말은 함수형 언어들의 구문적 표현은 다른 형태를 하고 있으나 이들의 의미는 사실 람다 식으로서 "단지 기호를 바꾸어 쓴 것에 불과하다"는 것을 뜻한다. 이것은 크게 틀린 말은 아니지만, 람다 계산법과 함수형 언어 사이에는 분명 차이점이 존재하며, 이것을 정확하게 설명하기 위해서 다음과 같은 관련 이론들을 소개한다.

람다 계산법과 함수형 언어의 근본적인 차이점으로서, 함수형 언어는 함수를 나타내는 기호 (function symbols)를 사용하고 있으나 람다 계산법에서는 그렇지 않다는 점이다. 앞서 언급한 사칙연산 더하기의 경우, 함수형 프로그래밍에서는 함수 기호를 사용하고 있으나, 람다 계산법에서는 이런 기호를 사용하지 않는다.

함수에 대한 기호를 사용하는 대표적인 정형시스템으로서 SML이나 Haskell과 같은 함수형 언어를 고려할 수 있으나, 이들보다 더 간결하고 많은 이론적 연구 배경을 갖춘 것으로서 항 개서 시스템 (term rewriting system)이 있다. 항 개서 시스템을 구성하는 가장 기본적인 요소는 변수, 상수 (인수를 갖지 않는 함수), 함수 기호들이며, 이들을 이용하여 항 (term)을 만들 수 있다. 그리고, 두 개의 항 l과 r이 주어졌을 때 이들을

$$l \rightarrow r$$

형태로 표현함으로써 룰을 정의할 수 있다. 여기서 $\rightarrow$ 관계는 반사적(reflexive)이며 추이적(transitive)인 성질을 갖는 것으로서, 직관적으로 이야기하면 등식 기호 (=)에서 symmetric한 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. 람다 계산법에서는 객체가 람다 식 (lambda term)인 반면에, 항 개서 시스템에서의 축약이 적용되는 객체는 항 (term)인 셈이다.

항 개서 시스템 중에서 한 독특한 형태를 취하고 있는 것으로서 적용 항 개서 시스템 (applicative term rewriting system)이 있다. 이 시스템의 함수 기호는 단 하나의 이진 (binary) 함수 `Ap`를 이용하고 있으며 나머지 모든 함수 기호들은 상수이다. 예를 들어, `plus (2, 3)` 형태의 텀을 적용 항 개서 시스템으로 표현한다면

```
Ap(Ap(plus, 2), 3)
```

으로 표현할 수 있다. 여기서 `plus, 2, 3` 모두 상수이다. 이것은 다음의 세 과정을 통해서 간략한 형태로 바꾸어 표현될 수 있다. 첫째, `Ap`를 중위연산자 형태의 기호 · 로 바꾸어 표현한다. 그러면,

```
((plus · 2) · 3)
```

을 얻을 수 있다. 두 번째 단계로, 이제 · 기호를 생략하기로 한다. 그러면

```
((plus 2) 3)
```

을 얻게 된다. 이제 마지막 단계로서 맨 바깥쪽 괄호를 생략하고, 또한 괄호를 좌측에서부터 적용(left associative) 한다는 룰을 적용하면 위의 항은

```
plus 2 3
```

을 얻는다. 이 세 과정은 단지 표현을 단순

히 하기 위한 방식으로서, 이 결과 표현되는 항을 원래의 항으로 복원하는 과정은 또한 쉽게 이루어 질 수 있다. 즉, plus 2 3이 주어졌을 때, ((plus 2) 3)과 ((plus ·2) ·3)의 과정을 거쳐서 원래의 항 Ap(Ap(plus, 2), 3)을 구할 수 있다.

적용 항 개서 시스템에서는 Ap를 제외한 모든 함수들이 상수이므로 함수 기호 뒤에 나오는 기호의 수에는 제한이 없다는 점이다. 즉, plus, Ap(plus, 2), Ap(Ap(plus, 2), 3), Ap(Ap(Ap(plus, 2), 3), 4) ... 등의 항들이 모두 합법적인 구문으로서, 이들을 위에서 소개한 간략한 형태의 항으로 표현하면,

```
plus
plus 2
plus 2 3
plus 2 3 4
 ...
```

등으로 표현된다. 이것은 plus가 상수임에도 불구하고 마치 인수를 하나도 갖지 않거나 고정되어 있지 않은 수의 인수를 갖는 함수인 것처럼 보이게 하는 착각을 하게 한다.

위에서 소개한 내용을 바탕으로, 어떤 한 항 개서 시스템의 항들을 적용 항 개서 시스템의 항으로 변환하는 함수를 currying이라고 한다. 예를 들어,

$$\text{plus}(2, 3) =_{\text{currying}} \text{plus } 2\ 3$$

으로 정의될 수 있다. 어떤 한 currying된 항을 uncurrying의 과정을 거쳐 원래의 항으로 복원될 수 있다.

$$\text{uncurrying}(\text{currying}(m)) = m$$

이 된다. Currying을 적용함으로써 구문 표현 시 괄호를 적게 사용할 수 있으며, 그 결과 항을 읽기가 편리한 장점을 얻을 수 있다. 여기서 논의된 내용의 정형적인 설명은 [3]에서 찾을 수 있다.

SML과 Haskell 등의 함수형 언어들은 적용 항 개서 시스템의 일종이다. 함수형 언어들은 적용 항 개서 시스템보다 더 복잡하고 룰의 적용에 있어서 특별한 규정을 적용하고 있기는 하지만 적용 항 개서 시스템의 한 부류로서 설명할 수 있다.

앞서 언급한 대로, 적용 항 개서 시스템에서는 함수가 갖는 인수의 수에 제약이 없는 느낌을 가질 수 있다. 이런 관점에서 항 개서 시스템 중에서 적용 항 개서 시스템은 람다 계산법에 좀 더 가깝다고 볼 수 있다. 적용 항 개서 시스템의 어떤 한 특정한 그룹들은 람다 계산법으로 번역될 수 있을 것으로 예상한다. 특히, 기호 Ap는 첫 번째 함수를 두 번째 함수에 적용시키는 역할을 하는 람다 식

$$\text{Ap} \equiv \lambda xy.\ x\ y$$

으로 표현할 수 있다. 즉 Ap는 람다 계산법의 β-레덱스를 생성하는 역할을 한다. 예를 들어, Ap(plus, 2)를 번역하면,

$$(\lambda xy.\ x\ y)\ \text{plus}_{\lambda}\ 2_{\lambda}$$

가 되어 결국

$$(\text{plus}_{\lambda}\ 2_{\lambda})$$

의 형태를 갖게 된다 (여기서 아래 첨자 λ가 붙은 것은 plus와 2를 각각 람다 식으로 번역한 것을 의미함). 모든 적용 항 개서 시스템이 람다 계산법으로 번역될 수는 없을 것이지만, 적어도 함수형 언어를 포함하여 적용 항 개서 시스템의 특정 부류는 람다 계산법으로 표현될 수 있다[4][5]. 이런 사실을 토대로 함수형 언어는 람다 계산법의 구문적 치장이라고 일컬을 수 있다.

지금까지 논의된 내용을 정리하면, 고계 함수의 개념은 람다 계산법과 적용 항 개서

시스템으로 표현되고 있다. 또한, currying을 통하여 일반 모든 함수의 구문을 적용 항 개서 시스템으로 표현할 수 있다. 함수형 언어는 적용 항 개서 시스템의 일종이다.

3. Fold 컴비네이터

3.1. 수와 재귀 함수

재귀 이론(recursion theory)과 그 응용은 컴퓨터 프로그래밍에서 가장 기본적인 기술이다. 앞서 논의한 대로, 명령형 프로그래밍의 루프 (loop)는 함수형 프로그래밍의 map이나 fold 등의 재귀적 특성을 갖는 함수로서 설명될 수 있다. 또한, 프로그래밍 언어와 형식 시스템의 구문 정의, 트리 등의 자료구조 정의에도 재귀적인 방법이 이용되고 있다. 본 장에서는 여러 형태의 재귀 함수 정의에 있어서 fold 컴비네이터를 활용하는 기술에 대해서 논의한다.

원시 재귀 함수 (primitive recursive functions) 이론은 0을 포함하는 자연수 에 대해서 정의되고 있다. 일반적으로 자연수에 대해 정의되는 재귀 함수 f는 다음과 같은 패턴을 가지고 있다.

```
f 0 = c
f (n+1) = h (f n)
```

자연수의 타입을 Nat이라 할 때, f, c, h는 f :: Nat -> a, c :: a, h :: a -> a의 타입을 갖는다.

자연수에 대한 데이터 타입은 구조적 귀납성 (structural induction)의 형태로 정의될 수 있는데, 이를 함수형 언어 Haskell로 표현하면 다음과 같다.

```
data Nat = Zero ¦ Succ Nat
```

즉, 자연수 Nat은 0 (Zero)이든가 혹은 Zero에 구성자 Succ을 반복적으로 적용한 것이다. 예를 들어, 1은 (Succ Zero)이며, 3은 (Succ (Succ (Succ Zero)))이다. Nat에 대해서 Peano의 사칙 연산 함수들은 다음과 같이 정의될 수 있다.

```
plus m Zero    = m
plus m (Succ n) = succ (plus m n)
mult m Zero = Zero
mult m (Succ n) = plus m (mult m n)
```

이밖에 계승 함수(factorial)나 피보나치(fibonacci) 함수도 유사한 방법으로 정의될 수 있다.

3.2. 재귀함수를 fold로 정의하기

fold는 함수형 프로그래밍에서 이용되고 있는 가장 대표적인 컴비네이터 중의 하나로서 모든 원시 재귀 함수 및 많은 재귀 함수의 정의에 이용될 수 있다. 재귀 함수 이론은 자연수 외에 리스트를 이용하여 표현할 수도 있다. Haskell에서 fold 함수는 수가 아닌 리스트에 대해서 정의되어 있으므로, fold 대신 자연수에 대해서 정의되는 함수 foldn을 다음과 같이 정의한다.

```
foldn g init Zero    = init
foldn g init (Succ n) = g(foldn g init n)
```

foldn는 입력 인수로서 재귀 함수 g와 자연수 Nat을 받는다. 각 재귀 함수에 대하여 그 함수의 init 또한 인수로서 정의된다.

앞서 언급한대로, 원시 재귀적 함수 f는

```
f 0 = c
f (n+1) = h (f n)
```

과 같은 패턴을 가지고 있다. 각 재귀 함수의 특성에 따라 c와 h가 결정된다. c와 h를 제외한 나머지는 프로그래밍 패턴은 공통적인데, 이와 같은 공통적인 패턴은 foldn 컴비네이터로서 표현될 수 있다. 즉, 모든 원시 재귀 함수는 위에 정의된 foldn 함수를 이용하여 표현될 수 있다[6].

일반적으로 위에 언급된 재귀함수 f를 foldn 으로 표현하는 것은

```
f = foldn h c
```

의 방정식에 대한 해답을 구하는 문제로서, 그 해답은 각 함수 f에 대한 적절한 h와 c를 찾는 것이다.

예를 들어, 앞서 정의된 Peano의 사칙 함수 plus와 mult를 foldn 함수를 이용하여 다음과 같이 재정의 할 수 있다. 여기서 유의할 점은 이진함수 plus와 mult 대신 위의 함수 패턴 f에 맞도록 (plus m)과 (mult m)의 unary 함수의 형태로 정의되어야 한다.

```
plus' m   =  foldn Succ m
mult' m   =  foldn (plus m) Zero
```

여기서 (plus m) 에 대한 h와 c는 각각 Succ 과 m이며, (mult m) 에 대해서는 각각 (plus m)과 Zero이다. 지수 함수 또한

```
expn' m = foldn (mult m) (Succ Zero)
```

로서 표현될 수 있다.

3.3. 고급 수준의 함수 표현

좀 더 복잡한 형태로 정의되는 재귀함수를 표현하기 위해서 다음과 같은 Ackmann 함수를 고려하자.

```
ack Zero  y     = Succ y
ack (Succ x) Zero = ack x (Succ Zero)
ack (Succ x) (Succ y)
              = ack x (ack (Succ x) y)
```

이 함수를 foldn으로 정의할 때는 두 단계를 걸친다. 이 함수의 인수의 특징에 따라 이 함수를 두 가지 패턴으로 나누어 생각한다.

(1) 첫 번째의 경우

```
ack Zero        = Succ
ack (Succ x)    = f (ack x)
```

이때 ack = foldn f v의 형태를 갖게 되는데, f에 대한 답을 구하기는 어렵지만 v = Succ 임을 쉽게 알 수 있다. f는 마지막 단계에서 구할 수 있다.

(2) 두 번째 경우

```
ack (Succ x)  Zero   = w
ack (Succ x)  (Succ y)
                  = g (ack (Succ x) y)
```

이 경우 ack (Succ x) = foldn g w 의 패턴으로 정의될 수 있으며, ack 룰에서 w = ack x (Succ Zero) 임을 쉽게 알 수 있다. g는 다음과 같이 얻을 수 있다.

```
ack (Succ x) (Succ y)
      =   g (ack (Succ x) y)
{ack룰을 적용하여}
ack x (ack (Succ x) y)
     =  g (ack (Succ x) y)
{(ack (Succ x) y)을 z 놓기로 하자}
ack x z   =   g z
```

이것은 곧 g = ack x임을 의미하며, 결과적으로,

```
ack (Succ x)
```

```
  =   foldn (ack x) (ack x (Succ Zero))
```

가 성립한다.

이제 (1)에서 언급한 f를 구하기로 한다.

```
ack (Succ x)   =   f (ack x)
```

{바로 위해서 구해진 결과를 적용하여}

```
foldn (ack x) (ack x (Succ Zero))
          = f (ack x)
```

{(ack x)를 h라 놓으면}

```
foldn h (h (Succ Zero)) = f h
```

{함수 f를 Haskell의 람다 식을 이용하여 표현하면}

```
f   =  \h -> foldn h (h (Succ Zero))
```

지금까지의 내용을 조합하면, ack는 다음과 같이 foldn를 이용하여 표현될 수 있다.

```
ack = foldn f v
= foldn (\h -> foldn h (h (Succ Zero)))
Succ
```

f x = g x 인 경우, f = g가 되는 현상을 람다 계산법에서 확장 원리(extentionality)라고 하는데, 확장원리는 고계함수가 아니면 성립될 수 없다. 따라서 재귀 함수를 foldn 컴비네이터로 표현하기 위해서는 구문이 반드시 적용 항 개서 시스템의 형태로 정의되어야 하는 것이 필수적이며, 일반 항 개서 시스템 형태의 함수형 언어라면 이것을 foldn로 정의하기는 어렵다.

계승 값을 계산하는 함수 fac과 피보나치 함수 fib은 다음과 같이 정의된다.

```
fac Zero  = Succ Zero
fac (Succ n) = mult (Succ n) (fac n)

fib Zero  = Zero
fib (Succ Zero)  = Succ Zero
fib (Succ (Succ n)) = plus (fib (Succ n))
(fib n)
```

이 두 함수를 foldn으로 정의하기 위해서는 이들을 f의 기본 패턴 형태로 변형시키는 것이 필요하다.

```
f 0 = c
f (n+1) = h (f n)
```

fac 함수의 경우 h는 (* (n+1))의 형태의 모습으로 표현되어야 하는데, 문제는 n이 좌변에 등장하는 변수이므로 h를 (* (n+1)) 형태로 표현하는 것은 불가능하다 점이다. fib 함수도 같은 문제를 갖게 된다. 이런 형태의 함수를 foldn으로 표현하기 위해서는 두 쌍으로 구성되는 튜플을 이용한다. fac 함수는 fac (n+1)을 계산할 때, 그전 단계에서 계산한 값 fac n을 이용하고 있다. fib 또한 같은 패턴으로 정의된다. 이런 현상은 튜플을 이용하여 표현할 수 있다.

fac을 튜플을 이용하여 계산할 때 사용되는 함수 f를 다음과 같이 정의한다.

```
f (a, b) = (Succ a, mult (Succ a) b)
```

이 함수 f는 튜플의 첫 번째 원소에는 자연수의 값을 하나씩 증가시키도록 하고 있으며, 두 번째 원소에는 전 단계에 계산된 결과 값 b를 이용하여 새로운 값 (mult (Succ a) b)를 계산하도록 한다. 즉, b는 어떤 자연수 n에 대한 (fac n)의 결과를 의미하며, (mult (Succ a) b)은 ((n+1)*(fac n)), 즉 (fac (n+1))에 대한 계산 과정을 수행하고 있다. 이와 같이, 튜플을 이용하여 어떤 한 단계의 계산을 할 때 전 단계의 계산 결과를 이용할 수 있으므로, 계산 속도도 빨라지는 장점을 갖는다. foldn을 이용하여 fac은 다음과 같이 새롭게 정의될 수 있다.

```
fac' = snd . (foldn f (Zero, Succ Zero))
     where f(m,n)
           = (Succ m, mult(Succ m) n)
```

여기서, foldn을 수행한 결과 값은 튜플 형

태로 되어 있으므로, 이 중에서 두 번째 값을 선택하는 snd 함수를 적용함으로써 최종 결과 값을 튜플이 아닌 자연수의 형태가 되로록 한다. 또한, 초기 값으로 fac (0) = 1에 해당하는 튜플 (Zero, Succ Zero)을 이용하고 있다. foldn은 함수 f가 반복적으로 수행될 수 있도록 하는 역할을 한다.

피보나치 함수 또한 foldn을 사용하여 다음과 같이 정의될 수 있다.

```
fib' = fst . (foldn f (Zero, Succ Zero))
      where f (m, n) = (n, plus m n)
```

이 함수는 fib (0) = 0, fib(1) = 1에 해당하는 두 초기값을 튜플 (Zero, Succ Zero)로서 정의하고 있다. 튜플 (a, b)에서 a = fib(n), b = fib(n+1)의 관계가 성립되며, 현 단계의 계산은 그 두 전 단계에서 계산된 결과를 이용하여 (a + b)를 수행함으로써 이루어지고, 이 값을 다음 단계 값을 계산하는데 이용하는 것이 필요하다. 위에 fib'의 f는 이런 방식으로 정의되었다.

3.3. 리스트 함수의 표현

앞서 자연수에 대한 재귀 함수를 fold 함수로 정의하는 방법과 이들을 구현하기 위해 foldn을 정의하였다. 리스트에 대해 정의된 재귀 함수들 또한 fold로서 표현될 수 있다. 구체적인 내용은 [7]을 참조하기 바라며, 여기에서는 많이 알려진 리스트에 대한 재귀함수들을 fold를 이용하여 프로그래밍한 예들을 소개한다.

```
fold f v []     = v
fold f v (x:xs) = f x (fold f v xs)

sum'  :: [Int] -> Int
sum'  = fold (+) 0
product' :: [Int] -> Int
product' = fold (*) 1
and' :: [Bool] -> Bool
and' = fold (&&) True
or' :: [Bool] -> Bool
or' = fold (||) False
(+++) :: [a] -> [a] -> [a]
xs +++ ys = fold (:) ys xs
length' :: [a] -> Int
length' = fold (\x n -> 1 + n) 0
reverse' :: [a] -> [a]
reverse' = fold (\x xs -> xs ++ [x]) []
map' :: (a -> b) -> [a] -> [b]
map' f  = fold (\x xs -> f x : xs ) []
filter' :: (a -> Bool) -> [a] -> [a]
filter' p  = fold (\x xs -> if p x then x
: xs else xs ) []
compose  :: [a -> a] -> a -> a
compose  = fold (.) id
dropWhile' :: (a -> Bool) -> [a] -> [a]
dropWhile' p  = fst . fold f v
 where f x (ys, xs)
       = (if p x then ys else x:xs, x:xs)
       v = ([],[])
```

4. 컴비네이터 프로그래밍 응용

4.1. 모나드 컴비네이터

함수형 프로그래밍의 기본 원리인 Church-Rosser 특성을 유지하면서 부대효과 (side-effect) 및 상태 제어 (state control)의 특성을 갖는 프로그래밍을 작성하는 것은 쉬운 문제가 아니다. 모나드가 이 문제를 해결하고 있다.

모나드에서는 return과 바인드 오퍼레이

터 (>>=)를 사용하고 있는데 이들은 계산(computation)을 제어하는 컴비네이터이다. 이 컴비네이터의 특징은 프로그래밍의 계산 과정을 제어할 수 있다는 점이다.

두 개의 프로그램 부분 A와 B가 있다고 가정하자. 이 둘을 합성함에 있어서 이 둘의 수행 과정을 결정하는 다음과 같은 세 가지 상황을 고려할 수 있다. 첫째, A와 B의 수행에 아무런 제약 없이 자유롭게 진행한다. 즉, A나 B를 임의로 선택하여 수행한 다음 나머지를 수행하거나, 이 둘을 병렬로 수행한다. 둘째, A와 B를 순서적으로 (sequencial) 수행한다. 즉, A-B나 B-A의 순서로서 수행한다. 셋째, A와 B를 선택적으로 수행한다. 예를 들어, A를 먼저 선택하여 수행하다가 만약 A의 수행결과가 예상한대로 진행되지 않는 경우 B를 수행한다. 이런 세 가지 상황에 대한 해결책은 다음과 같다. 첫 번째의 경우는 이미 함수형 언어의 특성에 내포되어 있으므로 새롭게 논의할 필요가 없다. 두 번째의 경우는 모나드의 바인드 오퍼레이터를 정의함으로써 해결할 수 있다. 세 번째의 경우는 모나드의 mplus 오퍼레이터를 정의함으로써 해결할 수 있다. 모나드의 구체적인 내용은 [9]를 참조하기 바란다.

4.2. 스크립트 프로그래밍

고계 함수 기능은 함수들을 합성하는 데 매우 유용함을 앞서 보여주었다. 이때 합성되는 합성들이 반드시 Haskell로 구현된 것일 필요는 없다. 즉, 인터페이스를 제대로 구현하기만 한다면 다른 언어로 구현된 함수들도 합성할 수 있다. 마이크로 소프트 사의 COM에서는 시스템에서 요구하고 있는 인터페이스 - 일반적으로 인터페이스 정의 언어(Interface Definition Language)로 기술됨 - 를 만족시키기만 한다면 서로 다른 언어로 개발된 함수들이라도 함께 합성할 수 있도록 하는 환경을 제공하고 있다.

이미 Haskell을 COM의 IDL과 인터페이스 시키는 라이브러리인 H/Direct 및 여러 관련 라이브러리들이 개발되어 시험 중에 있다[10]. 따라서 Haskell은 기존의 C, Java 등으로 구현된 함수들을 합성할 수 있게 되었다. 고계 함수 기능의 덕택으로, Haskell은 Tcl/Tk, Unix Shell 등의 기존 스크립트 언어보다 훨씬 우수한 스크립트 기능을 갖고 있다. 향후 함수형 프로그래밍 언어 기술이 스크립트 언어 설계 개발에 중요한 역할을 할 것으로 기대한다.

5. 결론

고계 함수는 함수의 기능을 세밀하게 분리하여 정의하고 이들을 합성할 수 있도록 함으로써 프로그램의 재사용을 극대화시킬 수 있다. 고계 함수의 원리는 람다 계산법과 적용 항 개서 시스템으로 설명할 수 있다.

함수형 프로그래밍에서는 고계 함수 기능을 이용한 다양한 컴비네이터들이 정의되어 사용되고 있다. fold는 재귀 함수의 기본 패턴을 정의하는 컴비네이터이다. 대부분의 재귀 함수들은 fold를 이용하여 재 정의될 수 있다. 즉, 다양한 재귀 함수의 정의에 fold를 재사용할 수 있음을 Haskell 프로그래밍을 통하여 제시하였다.

함수는 함수형 언어는 물론 기존의 C나 Java등의 명령형 언어를 이용하여 구현될 수도 있다. 마이크로소프트의 COM의 IDL 인터페이스 라이브러리가 개발됨에 따라,

Haskell 컴비네이터는 타 프로그래밍 언어로 구현된 함수들을 합성할 수도 있다. 고계 함수 기능의 프로그램 합성 기능과 타입 시스템 기능 덕택으로 함수형 언어는 향후 강력한 스크립트 언어로 발전할 수 있을 것으로 전망한다.

감사의 글

이 논문은 2000년도 학술진흥재단의 지원에 의하여 연구되었음 (KRF-2000-003-E00248).

참고문헌

[1] H. Barendregt, *The Lambda Calculus : Its Syntax and Semantics*, North-Holland, 1984.

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

[3] J.W. Klop. Term rewriting systems, In Abramsky et al., editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

[4] Sugwoo Byun, Richard Kennaway, and Ronan Sleep. Lambda-definable term rewriting systems. *LNCS*, No. *1179*, Springer-Verlag, pp. 106-115, December 1996.

[5] S. Byun, J.R. Kennaway, F. de. Vries, V. van Oostrom, Separability and translatability of sequential term rewrite systems into the lambda calculus, (submitted to a journal).

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

[7] Graham Hutton, A tutorial on the universality and expressiveness of fold. (http://www.cs.nott.ac.uk/~gmh).

[8] P. Wadler. Comprehending Monads. In *Proc-eedings of Symposium on Lisp and Functional Programming*, pp. 61-78. ACM. 1990.

[9] 변석우, 모나드 프로그래밍 응용, 36-46쪽, *2000년 추계 합동 학술 발표 논문집*, 한국정보과학회 프로그래밍언어연구회, 2000년 9월.

[10] S. Finne, D. Leijen, and E. Meijer. H/Direct: a binary foreign language interface for Haskell. In Proc. of *ICFP'98*, 1998.

변석우

1976~1980. 숭실대학교 전자계산(학사).
1980.~1982. 숭실대학교 전자계산(석사).
1982.~1999. ETRI 책임연구원
1988~1994. 영국 University of East Anglia 전산학(박사).
1998.~현재 경성대학교 정보과학부 조교수.

관심분야는 rewriting system, 함수형 프로그래밍, 의미론, 정형 시스템 등.