함수형 프로그래밍 기술을 이용한 프로그램 합성 방법[†]

경성대학교 변석우* 한양대학교 도경구*
경북대학교 정주희 동덕여자대학교 배민오

1. 서 론

오랜 전통과 많은 발전에도 불구하고, 프로그래밍 언어 기술은 산업체에서 아직 제대로 활용되지 못하고 있다. 산업체에서 주로 사용하는 C, C++, Java, Visual Basic 및 여러 스크립트 언어들은 프로그래밍 언어의 연구 결과를 충분히 반영하여 설계된 것이라고 볼 수 없다. 다른 분야와 마찬가지로, 프로그래밍 언어 분야에 있어서도, 학문적, 이론적 연구와 그들의 응용, 실용 사이의 긴밀한 연관성을 형성하는 것은 반드시 성취해야 할 중요한 과제이다. 최근 함수형 언어 분야에서는 우수한 이론을 바탕으로 개발된 함수형 언어를 여러 응용분야에 실용적으로 적용하려는 시도가 활발히 전개되고 있다. 본 고에서는 이에 대한 관련 기술 및 주요 동향에 대해서 소개한다.

함수와 그 관련 이론은 수학의 기본 이론으로서 이미 오래 전에 정립되어 널리 사용되고 있다. 프로그래밍언어에 있어서도, 거의 모든 언어들이 함수의 개념을 적용하고 있으므로 포괄적인 의미에서 이들을 모두 함수형 언어라고 볼 수도 있을 것이다. 또한, 함수형 언어의 종류와 특징이 다양하므로 함수형 언어에 대한 정의는 학자들 사이에서도 조금씩 다르다 [1]. 본 고에서는 함수형 언어에 대한 정의 대신, 우리가 논의하려는 함수형 언어의 특징을 구체적으로 제시한 후 이를 중심으로 논의를 계속하기로 한다.

최신 함수형 언어들의 대부분은 람다 계산법(lambda calculus)의 이론을 기반으로 설계 구현되고 있는 특징을 갖고 있다. 람다 계산법은 1930년대 Church에 의해 개발된 이래 확고한 이론적 정립과 함께 많은 발전을 이룩해 오고 있다. 특히, 1960년대부터 전산학의 프로그래밍 언어와 연관되면서 더욱 활발한 연구가 진행되어 오고 있다. 1970년대, Milner가 타입을 갖는 람다 계산법(typed lambda-calculus)에 대한 이론 연구와 함께 이를 응용한 함수형 언어 ML을 개발한 것은 프로그래밍 언어 역사의 중대한 발전으로 평가받고 있다[2, 3]. ML을 설계할 때, ML 계산 원리의 타당성과 주요 특징들은 람다 계산법의 환경에서 검증되었으며, 그 결과 ML은 기존 언어보다 더욱 추상화되고 명료한 의미(semantics)를 갖게 되었다. 이 방법은 그 후에 개발된 Miranda와 Haskell에 더욱 철저하게 적용되어, "함수형 언어는 람다 계산법의 문법적인 치장(syntactic sugaring)"이라고 불리고 있다(Lisp은 람다 계산법의 영향을 받기는 하였지만 그 정도에 있어서는 많은 차이가 있다).

타입을 갖는 람다 계산법을 기반으로 한 함수형 언어들은 모두 HOT(higher-order and type, 고차 함수와 타입)의 특징을 갖는다. 이 특징은 기존의 다른 언어들에서는 찾아볼 수 없는 것으로서, 함수형 언어와 다른 언어를 구분할 때 중요한 기준이 된다. 본 고에서는 이 특징에 대해서 간략히 소개하고, 이 특성을 이용한 프로그램 구성(혹은, 합성) 방법에 대해서 설명한다. 또한 함수형 언어를 산업체에서 적용한 성공 사례 Erlang에 대해서 소개한 다음, 함수형 언어의 향후 전망에 대해서 논의한다. 본 고에서 소개하는 함수형 프로그램의 예는 Haskell[4]로서 표현된다.

2. 함수형 프로그래밍의 특징

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

2.1 고차 함수

오늘날의 수학에서는 인수의 수(arity)가 고정되어 있는 형태의 함수를 이용하고 있다. 예를 들어, 더하기 함수 **plus**는 반드시 두 개의 인수를 갖도록 **plus(x, y)** 형태로 정의며, 그 타입은 **plus :: (Int, Int) -> Int** 형태로 표현된다. '함수를 데이타를 받아서 처리하는 기계' 라는 측면에서 볼 때, **plus**는 두 정수 값의 데이타를 동시에 입력받아, 결과 값으로 정수를 출력하는 기능을 한다.

그러나, 1930년대 람다 계산법을 연구하는 수학자들은 약간 다른 개념의 함수를 연구하였다. 람다 계산법에서의 함수는 반드시 '한번에 한 개의 인수' 를 처리하도록 되어 있다. 따라서, **plus** 함수는 첫 번째 정수 데이타를 입력받아 처리한 다음, 다시 두 번째 정수 데이타를 입력받아 처리한다. 그러면, 첫 번째 정수 데이타를 입력받아 처리한 결과 값은 무엇이 될까? 이 결과 값은 함수로서 표현된다. 이 현상은 다음과 같이 프로그래밍으로 표현할 수 있다.

```
plus :: Int -> (Int -> Int)
plus x y = plus1 x y
plus1 a = plus0 a
plus0 = (+)
```

plus의 타입은 **plus :: Int -> (Int -> Int)**로 표현된다. 이 의미는 하나의 정수 값이 입력되었을 때, **plus**는 그 결과 값으로 **(Int -> Int)** 형태의 '함수를 출력'한다. 즉, **(plus 2)**의 타입은 **(plus 2) :: Int -> Int** 으로서, 이것은 하나의 정수 값을 입력받아 정수 값을 출력하는 unary 함수임을 의미한다. 따라서, **(plus 2)** 함수에 새로운 값 **3**을 입력한 **((plus 2) 3)** 형태의 타입은 **((plus 2) 3) :: Int**의 타입을 갖는다. '괄호는 왼쪽부터 적용된다 (left-associative)'는 약속 하에, **((plus 2) 3)**는 괄호가 생략된 **plus 2 3**의 형태로 표현할 수 있으며, 그 계산 과정은 <u>**plus 2 3**</u> = <u>**plus1 2**</u> **3** = <u>**plus0**</u> 2 3 = (+)2 3 = 5이 된다 (함수형 프로그래밍에서 '계산 가능한 식'을 레덱스(reducible expression)라고 부는데, 여기서는 레덱스를 밑줄을 그어 표시하였다). 이와 같은 방식으로 표현되는 함수를 람다 계산법의 발전에 많은 공헌을 한 Haskell Curry의 이름을 따서 커리 함수 (Curried)라고 부른다. 최신 함수형 언어들은 대부분 커리 함수의 개념을 채택하고 있다.

이 커리 함수의 특징은 입력 데이타를 한번에 하나씩 처리한다는 점, 그리고 입출력 데이타로서 숫자 형태뿐만 아니라 함수 역시 입출력 값이 될 수 있다는 특징을 갖고 있다. 일반적으로 프로그램의 개념을 수동적인 '데이타'와 능동적인 역할을 하는 '함수'(혹은 '프로시저')로서 구분하는 경향이 있으나, 커리 함수에서는 이러한 이분법적인 개념은 존재하지 않는다. 숫자, 부울 값 등 우리가 일반적으로 생각하는 데이타는 인수가 0인 함수(즉, 상수)이다. 함수형 프로그래밍에서의 '값(value)'은 숫자나 True, False 등뿐만 아니라, 함수도 포함한다.

기존 프로그래밍에서는(인수가 제공되지 않는) 함수를 복귀 값이나 다른 함수의 인수로서 이용할 수 없는 제약이 있지만, 고차 함수를 이용하는 함수형 프로그래밍에서는 이러한 제약이 없다. 함수형 프로그래밍에서 함수를, 기존 프로그래밍의 수처럼, 다른 함수의 입력으로 사용할 수 있으며, 출력 값으로 이용할 수 있고, 저장할 수도 있다. 예를 들어, **[plus 3, fac, square]**는 하나의 정수를 입력하여 정수를 출력하는 unary 함수들을 리스트에서 저장한 경우로서, 이에 대한 타입은 **[Int -> Int]**이다. 함수형 프로그래밍에서는 함수가 위치할 수 있는 장소에 어떤 제약이 없이 자유롭다는 뜻에서 함수를 '일등 시민(first-class citizen)'이라 부른다. 또한 이론적으로 이런 형태로 표현될 수 있는 함수는 '고차 함수(higher-order function)'이라 일컫는다. 이러한 고차 함수의 기능은 프로그램 합성에 결정적인 역할을 한다.

커리 함수와 고차함수의 개념은 서로 밀접하게 연관되어 있으며, 모두 람다 계산법의 원리를 따른 것이다. 커리 함수에 대해 좀 더 자세한 설명을 원하는 독자는 [5]의 적용 항 개서 시스템(applicative term rewriting systems) 부분을, 람다 계산법에 대해서는 [6]을 참조하기 바란다.

2.2 타입 시스템

다른 언어들과 마찬가지로, 함수형 언어의 타입은 컴파일 과정에서 작성된 프로그램의 일관성을 점검하는 정보로서 이용된다. 그러나, 함수형 언어는 다른 언어보다 훨씬 더 풍부한 타입을 표현할 수 있다.

함수형 언어의 타입에서 가장 독특한 점은 타입의 정의에 변수를 사용할 수 있는 점이다. 예를 들어, 두 개의 원소로 구성된 튜플 중에서 첫 번째 원소를 선택하는 **함수 fst(x, y) = x**의 타입은 **fst :: (a, b) -> a** 이다. 여기서 a와 b는 타입 변수를 의미한다. 타입 변수는 상황에 따라 특정 타입으로 실례화(instantiation) 될 수 있다. 예를 들어, **fst(1, True) = 1**일 경우, **a**가 **Int**, **b**가 **Bool**로 실례화되어 **fst ::(Int, Bool) -> Int**의 타입을 갖게 된다. **fst("abc", [1,2]) = "abc"**로서 계산되는데, 이때 **fst**의 타입은 **fst :: (String, [Int]) -> String** 이다. 타입 변수를 사용함으로써 **fst**는 두 개의 원소로 구성된 모든 타입의 튜플들을 처리할 수 있다.

이와 같이, 타입 변수를 사용하면 변수의 실례화 기능에 따라, 함수가 여러 타입의 데이타들을 처리할 수 있으므로, 타입 변수를 사용하지 않는 경우보다 훨씬 더 '일반적(generic)'이다. 이는 곧 타입 변수에 의한 함수의 다형성(polymorphism)의 의미를 제시하고 있다. 이러한 다형성은 기존의 객체지향 프로그래밍에서는 이용할 수 없는 함수형 프로그래밍 고유의 중요한 기능이다. 또한, 함수를 정의할 때, 프로그래머는 타입을 정의할 수도 있고, 정의하지 않을 수도 있다. 만약 타입을 정의하지 않는 경우, 정의된 함수의 문맥(context)를 고려하여, 언어 시스템은 그 함수에 대한 가장 일반적인 타입을 유추해 낸다.

타입 변수와 함께, 대수 타입(algebraic type)은 함수형 프로그래밍의 유용한 기능이다. 대수 타입은 프로그래머에게 이미 정의된 여러 타입을 이용하여 새로운 타입을 정의할 수 있도록 한다. C의 **struct**와 객체지향 프로그래밍의 클래스 등도 이런 기능이 있지만, 대수 타입은 타입 구성자(constructor)와 타입 변수를 사용하고, 타입을 재귀적으로 정의할 수 있는 고급 기능을 가지고 있다. 예를 들어, 이진 트리 구조를 표현하기 위한 타입 **Tree**와 이 트리 노드의 있는 내용을 in-order 방식으로 정렬하는 함수는 다음과 같이 표현될 수 있다.

```
data Tree a = Nil | Node a (Tree a) (Tree a)
inorder :: Tree a -> [a]
inorder Nil = []
inorder (Node x t1 t2) = inorder t1 ++ [x] ++ inorder t2
```

Tree의 정의는 = 의 오른쪽에 표현된다. 오른쪽에서 |는 or의 의미를 갖는다. **Node a (Tree a) (Tree a)**을 선택하는 경우, 트리는 노드, 왼쪽 서브트리, 오른쪽 서브트리의 세 부분으로 구성된다. **a**는 노드 값의 타입을 의미하고 첫 번째 **(Tree a)**는 왼쪽 서브트리, 두 번째 것은 오른쪽 서브 트리로서 볼 수 있다. 트리를 구성하는 노드 값의 타입을 특정한 타입으로 정의하지 않고 타입 변수 **a**로서 표현함으로써, **Tree**는 임의의 타입을 갖는 노드들로 구성된 이진 트리를 표현할 수 있다. **Tree**의 오른쪽 부분을 **Nil**로 선택하는 경우, **Nil** 다음에는 더 이상의 다른 정보가 없으므로 트리의 잎을 의미한다. 여기서, **Nil**과 **Node**는 모두 타입 구성자이며, **a**와 **(Tree a)**는 타입이다. 대수 타입은 맨 앞에 타입 구성자가 나타나고, 그 다음에 $(n \geq 0)$ 개의 타입이 뒤이어 위치하는 형태로 표현된다. 자료 구조에서 트리는 재귀적으로 정의되고 있으며, 이러한 재귀적 개념은 대수 타입에서 자연스럽게 표현될 수 있다. 위의 예에서, **Tree**의 정의 부분에 다시 **Tree**가 재귀적으로 나타나고 있음을 볼 수 있다. 트리에 대한 타입이 자연스럽게 표현됨에 따라 이 트리를 다루는 함수도 쉽게 정의될 수 있다. 위의 예에서, 트리를 in-order 방식으로 방문하는 함수 **inorder**는 **Tree**의 구성 방법의 경우에 따라 정의되고 있다.

타입 변수, 재귀적 정의 방식에 의한 타입 정의는 자료 구조를 매우 간결하고 자연스럽게 표현할 수 있게 한다. 위의 예는 라이브러리를 사용하지 않고 순수하게 Haskell의 문법에 의해서만 표현되고 있다. 이 내용을 C와 같은 언어로 프로그래밍하는 경우와 비교하면, Haskell 프로그램의 우수성을 쉽게 느낄 수 있을 것이다.

3. 컴비네이터에 의한 프로그램 합성 예

3.1 컴비네이터 프로그래밍의 원리

모듈화(modularization)는 프로그램 개발의 보편적인 방법이다. 개발하려는 프로그램을 여러 작은 모듈 단위로 개발함으로써, 프로그램의 개발, 테스트 및 검증을 용이하게 할 수 있다. 모듈화의 주요 과제는 프로그램을 모듈 단위로 분리하는 것과 이 모듈들을 통합(혹은, 합성) (glue 혹은 synthesis) 하는데 있다. 이 둘은 매우 밀접하게 연관되어 있다. 모듈의 합성 기술에 따라 프로그램을 모듈로서 분리하는 문제

가 결정되기 때문이다. 모듈은 프로그램의 부분(fragments)을 의미한다. 프로그램을 구성할 때는 한 프로그램 내에 그 기능의 중복되지 않도록 모듈을 정의하는 것이 중요한 데, 이것은 곧 프로그래밍 합성 능력과 연관되는 것이다. 여기서 함수형 프로그래밍 고차 함수 기능의 실용적인 중요성이 돋보이게 된다. 함수형 프로그램의 고차 함수 기능은 고도의 프로그램 합성을 가능케 하며, 그 덕택으로 함수형 프로그래밍에서는 코드의 중복을 방지할 수 있고 프로그램을 간결하게 표현할 수 있다[7].

예를 들어, 다음과 같은 C 프로그램을 고려해보자.

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

이 프로그램은 **for** 문을 사용하여 배열 **a**의 각 요소에 있는 정수에 **sqrt** 함수를 적용하는(apply) 프로그램이다. 이것을 유사한 의미의 **Haskell** 함수로 코딩한다면 **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 **a**에 있는 각 원소들에게 **sqrt** 함수를 적용하도록 한다. **map**은 함수 **f**와 리스트 형태로 된 데이터들 **(x:xs)**를 받아들여, 받아들인 함수가 데이터 각 원소에 적용되도록 하는 것으로서, 이 기능은 C 언어의 **for**문에 해당된다고 볼 수 있다. **map**의 **f**에 바인딩되는 것은 함수이므로 **map**은 인수로서 함수를 이용하는 고차 함수의 기능을 이용하고 있다. **map**과 **for**의 차이점으로서, **for**는 단순히 **for** 그 자체만으로는 표현될 수 없고 반드시 적용될 함수와 데이타가 함께 표현되어야 하며 - 위의 예에서는 배열 **a**와 함수 **sqrt** - 따라서 이러한 패턴의 프로그램이 나올 때마다 **for** 문을 반복해서 작성해야 한다는 점이다. 이에 비해서, 함수형 프로그래밍의 **map**은 한번만 정의되면 그것을 다시 정의할 필요 없이 재사용할 수 있으며, 따라서 코드의 중복을 피할 수 있고, 프로그램을 간결하게 표현할 수 있게 한다. 위의 경우, **sqrt**는 함수의 인수로서 전달되어 사용되는 데, **map**은 단지 **sqrt** 뿐만 아니라 모든 unary 함수를 입력받을 수 있다. 예를 들어, 리스트 **b**의 각 원소의 값을 3만큼 증가시키려 할 때는 이미 정의된 **map**을 재사용하여 단순히 **map (+3)** $[b_1, ..., b_n]$ 으로서 표현하면 된다. 이와 같이, 고차 함수 기능은 정교한 모듈의 정의 및 합성을 가능케 하여, 결과적으로 코드를 재사용할 수 있게 하고 프로그램을 간결하게 할 수 있도록 한다.

위의 **map**과 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 **fold** 가 있다.

합성의 패턴에 따라 다양한 형태의 컴비네이터들이 정의될 수 있다. 최근에 많은 관심을 모았던 것으로서 함수형 프로그램의 계산 과정을 제어하기 위한 모나드 컴비네이터 바인드 (>>=)가 있다[8]. Haskell과 같은 순수 함수형 프로그래밍에서 입출력 등의 흐름 제어(control flow)를 제어하는 것은 매우 어려운 문제이지만, 모나드 컴비네이터를 이용하여 명령형 프로그래밍과 같은 느낌으로 흐름 제어를 프로그래밍하는 것이 가능하다(최근, 모나드를 발전시킨 arrow 컴비네이터에 대한 연구가 진행 중이다).

각 응용 분야의 특성에 따라 다양한 형태의 컴비네이터들이 개발될 수 있다. 최근 컴퓨터 그래픽스 및 애니메이션, 음악 작곡, 로봇 제어 등에 대한 컴비네이터들이 연구되고 있다[9]. 본 고에서는 애니메이션과 금융 계약서 작성을 위한 컴비네이터에 대해서 간략히 소개한다.

3.2 애니메이션 컴비네이터 라이브러리 Fran

일반적으로 애니메이션 등의 멀티미디어 프로그래밍은 상부 구조의 사용자 인터페이스에서부터 하부구조에 있는 내용의 표현(presentation)에 이르는 모든 부분들을 프로그래밍해야 한다. 하부 구조의 구체적인 부분들을(low level details) 프로그래밍 하는 것은 단조롭고 지루한 작업이다. 애니메이션은 비연속적인 그림의 컷들을 시간 영역에서 연속적으로 투영하는 실시간적 특성을 가지며, 한 화면 내에는 여러 개체들이 표현되고 있으며 이들은 각각 독립적으로 시간에 따라 변화하며 때로는 서로 통신하고 협력하며(cooperative) 동작하는 동시성(concurrency)을 지니고 있다.

Microsoft 사의 그래픽스 연구 그룹과 Yale 대학교의 함수형 프로그래밍 연구 그룹은 이와 같은 문제

를 해결하기 위한 연구를 수행하였으며, 그 결과 Haskell 환경에서 동작하는 Fran(Functional Reactive ANimation)이라는 동영상 프로그래밍 지원 컴비네이터 라이브러리들을 개발하였다[10, 9]. 여러 컴비네이터 중에서 대표적인 컴비네이터는 **over**이다. **over**는 동영상으로 출력되는 두 함수를 입력받아 이 둘을 하나의 화면에 합성시켜 출력하도록 하는 기능을 가지고 있다.

예를 들어, 다음과 같이 애니메이션 프로그램들을 합성할 수 있다.

```
animation1 = moveXY wiggle 0 bitMap1
animation2 = moveXY 0 waggle bitMap2
animation3 = over animation1 animation2
```

moveXY는 **Fran**에서 정의된 함수로서 **moveXY wiggle waggle image** 의 형태를 갖는다. **wiggle**은 (x,y)로 된 좌표에서 x를 의미하며, **waggle**은 y를 의미한다. **wiggle**과 **waggle**은 시간에 따라 그 값이 변화하며, **image**는 비트 맵 이미지 파일이다. **animation1**에서는 **waggle**의 값이 0으로 고정되었으므로 그림을 좌우로 이동시키는 동영상 함수이며, **animation2**는 그림을 상하로 이동시키는 동영상 함수이다. 이 두 동영상 함수는 **animation3**에서 **over** 컴비네이터에 의해 간단하게 합성되어, 두 개의 동영상 이미지가 하나의 화면에서 동시에 동작하게 된다.

제대로 구성되지 못한 동영상 프로그램인 경우, 두 동영상 이미지를 하나로 합성하기 위해서는 하부구조의 구체적인 것을 고려해야 하므로 문제가 복잡해진다. 그러나, **over**는 하부구조의 구현을 고려치 않고 매우 간단한 방법으로 동영상 이미지를 합성할 수 있도록 한다. **over** 컴비네이터의 구현을 가능케 하는 요인은 단순히 고차 함수의 기능뿐 아니라, Haskell의 대수 타입 및 타입 변수를 이용한 다형성도 함께 중요한 역할을 한다.

Fran은 멀티미디어가 함수형 프로그래밍에 적합한 응용 분야임을 보여 주고 있다. 앞서 언급한 멀티미디어 프로그래밍의 여러 지루한 문제들이 Fran과 유사한 형태의 컴비네이터 라이브러리와 함수형 언어를 통해서 매우 간결하고 추상적으로 처리될 수 있음을 예상할 수 있다. Fran 외에 음악 작곡 등의 다양한 형태의 멀티미디어 프로그램이 함수형 언어로서 구현되고 있다[9].

3.3 금융 계약을 위한 컴비네이터

금융 보험 분야의 계약서 작성에 함수형 언어의 컴비네이터 라이브러리가 매우 유용하다는 연구 보고가 발표되었다[11]. 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 위해서 함수형 프로그래밍의 컴비네이터 기술을 적용할 수 있다.

이 용도로 정의되는 컴비네이터는 금융 환경의 특성에 달려있다. 예를 들어, 가장 기본적인 형태의 계약인 **zcb**(zero-coupon discount bond) 함수 등을 정의한 후, 두 계약의 조건을 동시에 만족시키면서 새로운 계약을 구성하는 컴비네이터 **and :: Contract -> Contract -> Contract**를 이용함으로써 계약을 합성시킬 수 있다. 계약채무자와 영수자의 역할을 바꾸기 위해서 **give** 컴비네이터가 정의되며, **and**와 **give**를 이용하여 **andGive c d = and c (give d)**와 같은 새로운 컴비네이터를 구성할 수 있다. 이들 컴비네이터의 설명을 위해서는 금융 보험 분야에 대한 설명이 함께 제공되어야 하므로 이 정도에서 논의를 멈추기로 한다. 더 자세한 설명을 원하는 독자는 [11]을 참조하기 바란다.

4. 통신 교환기용 함수형 언어 Erlang의 경험

통신 교환기를 개발할 때 수천만 라인의 방대한 프로그램을 개발해야 한다는 것은 잘 알려진 사실이다. 따라서 대규모 프로그램 개발 과정에서의 소프트웨어 생산성, 안정성 등의 문제가 발생한다. 스웨덴의 유명 통신 교환기 개발 회사 Ericsson의 경우, 그들의 AXE-10 교환기 개발에 약 6천만 라인의 C++, EriPascal, Plex 소스 프로그램이 제작되었다고 한다. Ericsson은 프로그램 테스트 과정에서 많은 어려움을 겪었으며, 개발 기간의 단축, 소프트웨어 생산성의 향상, 안정성이 강화된 소프트웨어 개발을 위한 여러 방안을 모색하였다. 특히, 프로그래밍 언어의 중요성을 인식하여 Prolog 등의 여러 프로그래밍 언어를 이용한 실험을 거듭한 결과, 그들은 자체적으로 함수형 언어를 개발하여 사용하기로 하였다. 이런 배

경으로 1980년대 통신 교환기용 함수형 언어 Erlang이 개발되었다[12].

Erlang은 교환기 시스템의 특성을 반영하여, 대규모 소프트웨어 개발에 적합하고, 실시간과 동시성의 기능을 지원이 가능하도록 설계되었다. 다른 언어로 개발된 기존 프로그램들을 함께 수행시키기 위한 여러 개발 환경을 갖추었다. 1980년대 후반부터 지금까지, Erlang은 통신 교환기 소프트웨어 개발에 꾸준히 사용되어 오고 있다. 언어의 기능과 성능도 계속 발전하고 있다. 처음에는 타입이 없는 언어로 시작하였지만, 후에 타입 기능이 보강되었으며, 많은 라이브러리들과 유틸리티들이 개발되었다. 지금까지 사용해 본 결과, 실시간, 속도, 소프트웨어의 생산성 등 주요 조건을 충분히 만족시킨다고 한다. C++와 비교할 때 1/7 정도 수준으로 소스코드의 양을 줄일 수 있었으며, 소프트웨어에의 에러가 거의 없으므로, 안전성이 강조되는 시스템 개발에 적합하다는 판단을 하고 있다. 이러한 성공을 바탕으로, Ericsson은 Erlang을 상품화하기에 이르렀다. 시스템 개발 도구로서 제작된 프로그래밍 언어가 독립적인 상품으로 발전하게 된 것이다.

현재 Erlang은 Ericsson 내에서 약 12 개 정도의 프로젝트에서 사용되고 있는데, 대형 프로젝트인 AXD301 ATM 교환기 시스템도 그 중에 하나이다. 이 프로젝트에는 약 300명 정도의 프로그래머가 참여하고 있으며, 지금까지 개발된 Erlang 소스코드의 양은 대략 85만 라인 정도라고 한다. 전체적으로 Erlang은 Ericsson 내의 35개 사이트와 외부의 80개 사이트에서 사용되고 있다.

Erlang의 성공 사례는 우리에게 매우 교훈적이다. 함수형 언어의 응용 분야는 인공지능 등 주로 학구적이거나 연구 분야에 집중되어 온 경향이 있지만, Erlang의 응용 분야는 그러한 응용 분야와는 매우 다르다. 우선, Erlang을 개발하게 된 목적이 소프트웨어의 개발비용을 줄이고 안전한 시스템을 개발하기 위한 현실적인 문제의식 위에서 출발하였다는 점에 주목할 필요가 있다. 일반적으로 소프트웨어 개발 시, 새로운 프로그래밍 언어 기술을 적용하려는 노력 없이, 언어의 선택을 과거의 관성에 의존하는 경우가 많다. 그러나 Ericsson은 응용 분야의 문제를 철저히 분석하고, 그에 적합한 언어를 선택하는 전략을 사용하여 큰 성공을 거두었다.

Erlang의 성공 요인은 단순히 언어 그 자체의 우수성에만 기인하는 것은 아니다. Erlang 설계자들은 응용 분야의 문제점과 의견을 적극적으로 수용하여, 개발자들이 요구하는 라이브러리, 기술 문서, 교육 등을 제공하였다. 이와 같이 언어 설계자와 응용 프로그램 개발자들 사이의 긴밀한 의견 교환 및 문제 해결 노력 역시 매우 중요한 요인으로 평가되고 있다.

5. 결론 및 향후 전망

프로그램 합성은 소프트웨어 개발의 근본적이고 보편적인 방법이다. HOT와 이를 기반으로 한 컴비네이터 기능을 갖는 함수형 언어는 프로그램 합성에 탁월한 우수성을 갖고 있다. 최근의 연구 결과를 통해서, 컴비네이터 기술은 동영상 프로그램, 음악 작곡, 로봇 제어, 금융 보험 계약 등 광범위한 응용 분야에 이용될 수 있음을 보여주었다. Ericsson의 Erlang을 이용한 대규모 소프트웨어 개발 경험은 함수형 언어 발전의 좋은 사례이다. 이런 사례들을 고려할 때 향후 함수형 언어를 산업체 응용 분야에 적용하려는 시도는 더욱 활발해 질 것으로 예상한다. 물론 해결해야 할 여러 문제가 있다고 보여 진다. 특히, 국내에서는 함수형 언어에 대한 경험이 매우 빈약한 상태이므로, 함수형 언어에 대한 관심과 교육의 장려가 우선되어야 할 것이다.

향후 Microsoft의 .net 환경은 함수형 언어를 포함한 새로운 언어의 실용화에 큰 도움을 줄 수 있을 것으로 전망한다. .net은 어떤 한 특정 언어를 위주로 하는 것이 아니라, 기계어 수준의 호환성에 의존하므로, 다중 언어를 지원할 수 환경을 갖추고 있다. 어떤 프로그래밍언어라도 그 컴파일러가 .net 환경에 포함되어 있으면, 그 언어로 개발된 컴포넌트를 다른 컴포넌트와 함께 통합시킬 수 있다. .net 환경에서 컴포넌트들은 어떤 특정 언어로 개발될 필요가 없으며, 새로운 프로그래밍언어라고 해서 기존 프로그래밍언어에 비해 불리하지도 않다. 그러므로, 함수형 언어와 같이 기존에 널리 사용되지 않는 언어의 실용화가 훨씬 쉬워질 수 있다.

컴포넌트 프로그래밍에서 컴포넌트의 외형적인 모습은 함수로서 보여 진다. 이 경우, 컴포넌트는 추상 데이타 타입(abstract data type) 형태의 구조를 갖게 된다. 외형적으로 보여지는 함수는 추상적인 사양(specification)이고, 이 사양을 구현하는 언어나 구체적인 구현 방법은 사양과 독립적이다. 따라서, 여

러 다른 언어 구현된 컴포넌트들이라고 하더라도, 이들의 스크립팅은 함수의 합성 문제이며, 따라서 컴비네이터 기능을 갖는 함수형 언어가 유용하다[13]. 이런 배경으로 Mondrian과 같은 함수형 스크립트 언어가 개발되고 있다[14].

참고문헌

[1] http://www.cs.nott.ac.uk/~gmh/faq.html(함수형 프로그래밍에 대한 FAQ).

[2] Robin Milner. A theory of type polymorphism in programming. Journal of Computer and System Science, 17, 348~75, 1978.

[3] Robin Milner. A proposal for Standard ML. Proc. of ACM Symposium on Lisp and Functional Programming, 1984.

[4] http://www.haskell.org(Haskell 언어 홈 페이지)

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

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

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

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

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

[10] http://conal.net/fran/tutorial.htm(Conal Eliott의 Fran 홈 페이지)

[11] 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)

[12] http://www.erlang.se(Erlang 언어 홈 페이지)

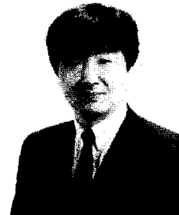
[13] Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM Components in Haskell, In Proc. International Conference on Software Reuse, 1998.

[14] http://www.mondrian-script.org(Mondrian 스크립트 언어 홈 페이지)

변 석 우

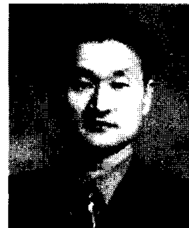
1976 숭실대학교 전자계산(학사)
1980 숭실대학교 전자계산(석사)
1982 ETRI 책임연구원
1988 영국 University of East Anglia 전산학(박사)
1998~현재 경성대학교 정보과학부 조교수
관심분야 : rewriting system, 함수형 프로그래밍, 의미론, 정형 시스템 등
E-mail : swbyun@star.kyungsung.ac.kr

도 경 구



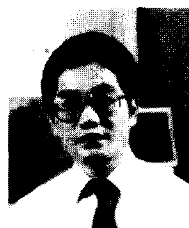
1976 한양대학교 산업공학과(학사)
1984 아이오와주립대 전산학(석사)
1987 캔사스주립대 전산학(박사)
1993~1995 Univ. of Aizu 교수
1995~현재 한양대학교 교수
관심분야 : 프로그래밍언어, 프로그램 정확성 및 안전성 검증, 스마트카드
E mail : doh@cse.hanyang.ac.kr

정 주 희



1979 서울대학교 요업공학과(학사)
1981 KAIST 재료공학과(석사)
1981~1984 동력자원연구소 연구원
1991 UC Berkeley, Dept. of Mathematics, Ph.D.
1991~1995 McMaster Univ., U. of Waterloo etc. 박사후 연구원
1997~현재 경북대학교 수학교육과 조교수
관심분야 : 논리 및 계산, 프로그래밍 언어 의미론, 대수적 모델 이론
E-mail : jhjeong@knu.ac.kr

배 민 오



1981 서울대학교 전자공학과(학사)
1983 한국과학기술원 전산학과(석사)
1983 삼성전자 종합연구소
1992 Syracuse University 전산학(박사)
1995 삼성 SDS 정보기술연구소
1996~현재 동덕여자대학교 정보과학대학 부교수
관심분야 : 논리프로그래밍, 정리증명이론, 데이타베이스설계, 전자상거래 시스템
E-mail : bai@dongduk.ac.kr