□ 기술해설 □

타입 시스템에 의한 체계적인 복합형 프로그래밍†

한양대학교 도경구* · 곽노건**

● 목 차 ●

1. 서 론

Hindley/Milner의 타입 시스템(type system)[1]은 타입 오류에 대한 안전성을 보장하고, 타입을 자동으로 유추하는 특징 이외에도 프로그램의 유연성(flexibility)을 체계적으로 높여주는 장점이 있다. 예를 들면, 정수 배열(array)을 정렬하는 C 언어 프로그램은 문자 배열을 정렬하는데는 쓰여질 수 없다. 왜냐하면 같은 알고리즘을 사용하는 유사한 프로그램일 지라도 C 언어는 하나의 타입만 취급할 수 있는 단일형(monomorphic)이기 때문이다. 따라서 하나의 프로그램이 어떤 타입의 값들도 모두 취급할 수 있도록 허용이 된다면 그 프로그램이 다목적으로 여러 가지 다른 유사한 상황에 두루 쓰여질 수 있기 때문에 경제적이고 체계적인 프로그램 개발이 가능할 것이다. 이와 같이 여러 가지 타입의 값들을 모두 취급할 수 있는 성질을 가진 프로그램을 복합형(polymorphic) 프로그램이라고 한다. 이렇게 복합형 프로그래밍을 가능하게 하는 언어로는 Standard ML, Haskell, Gofer 등 주로 함수언어(functional language)들이 있는데, 본 글에서는 이 언어들이 제공하는 복합형 프로그래밍에 관하여 살펴보고자 한다.

† 본 연구는 정보통신연구관리단 대학기초연구지원사업 과제(과제번호 : U96-172)의 부분적인 지원을 받았습니다.
*정 회 원
**학생회원

2. 복합형(Polymorphic) 프로그램

Hindley/Milner의 타입 시스템은 프로그램을 실제로 수행하지 않고도 프로그램의 각 부분에 대해 그 곳에 맞는 타입을 유추한다. 일단 적절한 타입이 유추되면 프로그램 수행시 절대로 타입오류가 발생하지 않는다고 보장할 수 있고, 만일 타입을 유추할 수 없게 되면 그 프로그램은 수행시 타입오류를 가지고 있다는 것을 의미하게 된다. C나 Pascal 같은 언어의 타입 시스템은 쓰이는 각 변수들의 타입에 관한 정보가 주어진 상태에서 타입을 검사하는데 반해서, Hindley/Milner 타입 시스템은 타입에 관한 정보가 주어지지 않더라도 타입을 자동으로 유추할 수 있다. 예를 들어서 0이 주어지면 True를 결과로 주고, 그렇지 않은 경우는 False를 주는 함수는 다음과 같이 정의가 된다.

```
isZero 0 = True
isZero n = False
```

위 함수의 경우 타입은 Int → Bool로 자동유추가 되는데, 의미는 Int 타입의 값이 주어지면 결과 타입은 Bool이 되고, 그렇지 않은 경우는

모두 타입 오류로 처리된다. 즉 isZero 3은 False를 결과로 주지만, isZero 3.0은 타입오류가 된다. 따라서 Int 이외의 다른 타입은 쓸 수 없도록 제한이 되고, 이런 함수를 단일형(monomorphic) 함수라고 한다.

반면에 여러 타입의 값에 대해서 균일하게 동작하는 함수를 복합형(polymorphic) 함수라고 한다. 복합형 함수의 예로서 다음과 같이 리스트의 길이를 계산하는 프로그램을 살펴보자.

```
length [] = 0
length (x : xs) = 1 + length xs
```

이 length 함수는 List a → Int 의 타입을 가지게 되는데, 여기서 a는 타입변수(type variable)를 나타내므로, 이 함수는 어떤 타입의 리스트에도 다 적용될 수 있다. 예를 들면 length [7,4,7]과 length ['s','k','y']는 모두 주어진 리스트의 타입에 관계없이 3을 결과로 준다. 또 하나의 복합형 함수의 예로서 map 함수를 살펴보자.

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

map은 고차원 함수로서 함수 f와 리스트 ls가 주어지면 그 리스트 ls의 모든 원소에 함수 f를 적용시킨 리스트를 결과로 주는 함수이다. 여기서 (x : xs)의 ':'은 리스트 결합자로서(보통 cons라 부름), x는 그 리스트의 첫 번째 원소이고, xs는 x를 제외한 나머지 리스트를 가리킨다. 이 map함수의 타입은 (a → b) → (List a → List b) 가 된다. 이해를 돕기 위하여 이 두 함수의 타입을 전칭기호(universal quantifier)를 사용하여 정식으로 표시하면 각각 ∀a. List a → Int 와 ∀a. ∀b. (a → b) → (List a → List b) 가 된다.

3. 타입이 중복(Overloading)된 프로그램

Hindley/Milner의 타입 시스템은 length와 같은 전천후 복합형 함수를 정의하기에는 적절하지만, 부분적으로 몇몇 원하는 타입에 대해서만 제한적으로 사용할 수 있는 함수를 정의하기는 불가능한 단점이 있다. 예를 들어서 2절의 isZero 함수는 정수 타입에 대해서만 결과를 주는 단일형 함수이다. 그러나 만일 실수 등의 다른 타입의 수에 대해서도 적용이 될 수 있도록, 즉 isZero 3 이나 isZero 3.0 모두 False 의 결과를 주도록 정의하고 싶은 경우에는 isZero 함수가 Int → Bool 타입과 Real → Bool 타입을 동시에 가질 수 있도록 허용하여야 한다. 그러나 기존의 Hindley/Milner의 타입 시스템은 가장 일반적인 타입 하나만을 유출하므로 두 타입을 동시에 가지는 것은 불가능하다. 정수와 실수를 동시에 처리할 수 있도록 하게 하는 유일한 방법은 isZero 함수가 a → Bool 타입을 갖도록 하면 되는데, 이 같은 경우 a는 타입변수로서 어떤 타입이 되어도 상관이 없으므로 안전하지가 않다. 이렇게 모든 타입이 아닌 제한된 일부 타입에만 복합(polymorphism)을 허용하는 경우를 타입이 중복(overloading)되었다라고 한다.

Standard ML의 타입 시스템은 일부 미리 지정된 연산자(operator)들에 한해서 중복을 허용한다. Standard ML의 + 나 - 연산자는 정수와 실수 덧셈에 중복하여 사용할 수 있다. 2개의 값을 비교하는 = 연산자의 경우 a * a → Bool 과 같은 복합형 타입을 갖도록 정의하면 이상적이다. 그러나, 함수언어에서는 함수도 값으로 취급되며, 함수끼리의 비교는 불가능하므로 a가 함수 타입이 되지 않도록 제한해야 한다. 따라서 = 연산자는 함수 타입을 제외한 모든 타입에 중복하여 사용할 수 있도록 정의될 수 있으면 좋다. Standard ML은 함수를 제외한 모든 값을 비교가능타입(equality type)으로 분류하여, 비교가능타입을 나타내는 타입변수는 일반타입변수와 구분하여 타입 변수 앞에 '를 붙임으로서 'a 또는 ''a 등으로 표시하여 구별을 한다. 따라서, =연산자의 경우 'a * ''a → Bool 타입을 가지게 함으로서 'a는 함수 타입이 아닌 모든 타입에 중복이 되게 한다. 그러나 이와 같이 Standard ML에 적용된 해법은 일부 미리 정해진 연산자들에 한정되어 적용시킬 뿐 타입을 체계적으로 중복시킬 수 있도록 하는 문제를 근본적으로 해결하지는 못했다.

4. Haskell의 타입 클래스(Type Class)

Haskell 언어[2,3,5]는 타입 클래스(type class)를 통하여 체계적으로 중복을 허용하는 함수를 정의할 수 있다. 타입 클래스는 여러 타입들의 집합인데, 미리 지정된 타입 클래스는 Eq, Ord, Num 같은 것들이 있다. Eq는 비교가 가능한 모든 타입들을, Ord는 순서를 매길 수 있는 모든 타입들을, Num은 연산을 할 수 있는 모든 타입들을 포함하고 있다. 여기서 Ord에 속하는 모든 타입과, Num에 속하는 모든 타입은 Eq에 속한다. 그러나 Ord와 Num에 동시에 속하는 Int 같은 타입이 있는 반면에, 둘 중 어느 한 클래스에는 속하나 다른 클래스에는 속하지 않는 Char 나 Fractional 같은 타입도 있다. 타입 클래스를 사용하면 + 연산자의 경우는 Num a => a→a→a 타입을 가지는데, 연산을 할 수 있는 Num에 속한 타입만을 허용한다는 제한을 명료하게 줌으로서 체계적으로 타입의 중복을 허용한다. 비슷하게 두 값을 비교하는 연산자인 ==의 경우는 Eq a => a→a→Bool 타입을 가진다.

Haskell의 타입 클래스를 이용하면 간략하고 매우 유연성 있는 프로그램을 작성할 수 있다. 예를 들어 나무(tree)구조를 생각해 보자. 나무(tree)구조에는 유사하지만 모양이 다른 여러 가지 나무구조들, 즉 이진나무(binary tree), 이진검색나무(binary search tree), 표찰이 붙은 나무(labelled tree) 등이 있다. 이렇게 각기 조금씩 다른 나무구조에 공통적으로 적용이 되는 함수들은 depth, size, paths, subtrees 등이 있다. 이러한 함수들이 Haskell의 타입 시스템을 이용하여 유사한 여러 나무구조에 중복되어 쓰일 수 있는지 살펴보자. 우선 간단한 이진 나무는 다음과 같이 정의 될 수 있다.

```
data BinTree a =
        Leaf a
      | Node (BinTree a) (BinTree a)
```

여기에서 이진나무구조 BinTree 타입을 가진 나무는 a 타입의 값을 가진 Leaf이거나, 왼쪽 가지(subtree)와 오른쪽 가지가 각기 이진나무구조를 가진 Node가 되게 회귀적으로(recursive)으로 정의된다. 예를 들면 Node (Leaf 2) (Node (Leaf 1) (Leaf 3))은 BinTree Int 타입을 가진 이진나무이다. 또한 이진검색나무(binary search tree)는 다음과 같이 잎을 포함한 모든 노드(node)가 a라는 타입의 값을 가지도록 정의된다.

```
data BSTree a =
        Empty
      | Split a (BSTree a) (BSTree a)
```

예를 들면 Split 2 (Split 1 Empty Empty) (Split 4 (Split 3 Empty Empty) Empty)는 BSTree Int 타입을 가진 이진검색나무이다. 여기에서 BinTree나 BSTree는 새로운 타입을 정의하므로 타입 생성자(constructor)라고 부르고, Leaf, Node, Empty, Split 들은 그 타입에 속하는 새로운 데이터를 정의하므로 데이터 생성자라고 한다. 위의 두 나무 구조는 같지는 않지만 유사하기 때문에 하나의 타입 클래스로 분류할 수 있고, 다음과 같이 정의된다.

```
class Tree t where
      subtrees :: t → [t]
```

여기서 Tree는 새로 정의되는 타입 클래스의 이름이고, where 이하는 이 클래스에 속하는 모든 타입에는 공통적으로 적용할 수 있는 함수들을 나열한다. 여기에서 subtrees 함수가 주어진 어떤 나무에 대해 그 나무의 가지들의 리스트를 결과로 주는 함수라면, 다음과 같이 각 나무 구조에 맞는 subtrees함수를 각각 정의할 수 있다.

```
instance Tree (BinTree a) where
    subtrees (Leaf n) = []
    subtrees (Node l r) = [ l , r ]
instance Tree (BSTree a) where
    subtrees Empty = []
    subtrees (Split x l r) = [ l , r ]
```

위의 정의들을 바탕으로 하여 Tree 클래스에 속하는 모든 나무 구조에 공통으로 쓰일 수 있는 중복 함수를 정의 할 수 있다. 예로서, 나무의 깊이를 계산하는 depth 함수를 살펴보자.

```
depth :: Tree t => t -> Int
depth=(1+) . foldl max 0 . map depth .
        subtrees
```

상당히 추상적이어서 함수형 프로그램을 많이 접해보지 못한 독자들에게는 생소해 보일지도 모르지만, 대강 의미를 설명해 보면 다음과 같다. "Tree 클래스에 속하는 타입을 가진 나무가 주어지면, 그 가지들의 depth를 각각 구해서 그 최댓값에다가 1을 더한 값을 그 나무의 depth로 한다." 위의 프로그램은 어떤 나무에도 중복 적용할 수 있는 일반적인 프로그램이라는 장점이 있다. 반면에 다음과 같은 이진 나무 구조에만 쓸 수 있는 함수에 비하면 효율적인 면에서 뒤떨어진다. 왜냐하면 계산 도중에 가지들의 리스트를 유지하는 비용이 더 들기 때문이다.

```
depth (Leaf n) = 0
depth (Node l r)=1+max(depth l, depth r)
```

그러나 특기할 만한 사실은 후자와 같은 전문화된 단일형 프로그램은 전자와 같은 일반적인 중복 복합형 프로그램으로부터 컴파일러에 의해서 자동으로 생성될 수 있다는 것이다.

5. 고차 Hindley/Milner 타입 시스템

2절의 Hindley/Milner 타입 시스템에 의하면 모든 타입의 값들에 대해서 일정하게 수행이 되는 복합형 함수를 정의할 수 있다. length는 어떤 타입의 리스트라도 길이를 계산할 수 있고, map은 주어진 함수의 입력 타입과 주어진 리스트의 요소 타입이 일치하는 모든 함수와 리스트를 처리할 수 있다. map의 타입을 정식으로 표현하면 ∀a. ∀b.(a→b)→(List a→List b)가 되는데, 기존의 Hindley/Milner 타입 시스템에 의하면 타입은 일반화하여 타입 변수로 표시될 수 있지만, List 같은 타입결합자는 일반화될 수 없다. 그런 의미에서 기존의 타입 시스템을 1차 타입 시스템이라고 한다.

반면, 타입결합자에 대해서도 일반화를 할 수 있다면, map 함수도 ∀t. ∀a. ∀b.(a→b)→(t a→t b)와 같은 타입을 가질 수 있을 것이다. 그렇게 되면 List뿐만 아니라 BinTree 등 다른 유사한 타입결합자에 대해서도 map 함수가 복합으로 쓰여질 수가 있어서, 더욱 일반적이고 유연성 있는 프로그램을 작성할 수 있을 것이다. 이와 같은 타입 시스템을 고차 타입 시스템이라고 하는데, 다음과 같이 종류(kind) 개념을 도입함으로서 형식화 할 수 있다[4]. 종류는 타입결합자의 집합이고, 종류의 구문(syntax)은 다음과 같이 정의된다.

$$\kappa ::= * \mid \kappa_1 \rightarrow \kappa_2$$

예를 들면 Int 나 Bool 같은 타입은 * 종류이고, 독립변수(argument)가 없는 타입결합자로 취급된다. List 나 BinTree 같은 타입결합자는 *→*종류에 속한다. 그리고 각 κ에 대해서 타입결합자는 다음과 같이 정의된다.

$$C^{\kappa} ::= x^{\kappa} \mid a^{\kappa} \mid C^{\kappa' \rightarrow \kappa} C^{\kappa'}$$

여기서 x^{κ}는 상수타입결합자를 나타내는데, Int 나 Bool들은 * 종류의 상수타입결합자이고, List 나 BinTree들은 *→*종류의 상수타입결합자이다. a^{κ}는 κ종류의 변수타입결합자를 나타낸다. 그리고 List Int 는 *→* 종류를 * 종류에 적용(apply)시키므로, 결과적으로 * 종류의 타입결합자가 된다.

6. Gofer의 결합자(Constructor) 클래스

Gofer 언어는 Haskell 계열의 언어로서 Haskell의 1차 타입 시스템을 고차로 확장시킨 결합자 클래스 (constructor class)를 사용하는 것이 특징이다[4]. Gofer에서는 결합자 클래스를 사용하여, map 함수가 List 타입결합자뿐 만 아니라 BinTree 등의 다른 타입결합자들에도 확대적용이 가능하게 한다. 예를 들면 다음의 Functor 라는 결합자클래스를 통하여, map 함수를 List 결합자뿐 아니라, *→* 종류인 다른 모든 결합자에도 복합하여 사

용할 수 있도록 확장할 수 있다.

```
class Functor f where
  map :: (a -> b) -> (f a -> f b)
instance Functor List where
  map f [] = []
  map f (x:xs) = f x : map f xs
instance Functor BinTree where
  map f (Leaf x) = Leaf (f x)
  map f (Node l r) = Node (f l) (f r)
```

이는 복합형 프로그램을 타입뿐만 아니라 타입결합자에도 자연스럽게 확장함으로서 더욱 일반화된 프로그램 작성을 가능하게 한다.

7. 결 론

본 글에서는 타입 시스템을 이용하여 복합 및 중복 프로그램을 체계적으로 작성할 수 있도록 지원해 주는 Haskell과 Gofer 언어와 그 이론적 배경에 대하여 간략하게 살펴보았다. 이 언어들은 상당히 효율성이 좋은 인터프리터/컴파일러가 이미 구현이 되어있고, 이를 이용하여 여러 응용 소프트웨어 개발이 활발히 진행되고 있다. Haskell과 Gofer 언어에 대해서 더 상세히 알고 싶은 독자는 참고 문헌과 http://haskell/systemsz.cs.yale.edu/haskell을 참조하기 바란다.

참고문헌

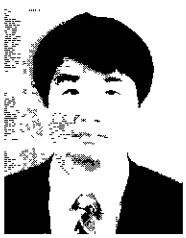
[1] L. Damas and R. Milner, Principal type schemes for functional programs, In *9th Annual ACM Symposium on Principles of Programming Languages*, pp. 207-212, Albuquerque, New Mexico, January 1982.

[2] P. Hudak and J.H. Fasel, A gentle introduction to Haskell, *ACM SIGPLAN Notices*, 27(5), May 1992.

[3] P. Hudak, S. Peyton Jones and P. Wadler (editors), Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2), *ACM SIGPLAN Notices*, 27(5), May 1992 [Haskell 1.3, http://haskell.cs.yale.edu/haskell-report.html].

[4] Mark P. Jones, Functional programming with overloading and higher-order polymorphism, Advanced Functional Programming, In *Proc. Bastad Spring School*, LNCS 925, May 1995.

[5] Simon Thompson, Haskell-*The Craft of Functional Programming*, Addison-Wesley, 1996.



도 경 구

1980 한양대학교 산업공학과, 학사
1987 미국 Iowa State University 전자계산학, 석사
1992 미국 Kansas State University 전자계산학, 박사
1993~95 일본 會津大學 교수
1995~현재 한양대학교 전자계산학과 조교수
관심분야 : 프로그래밍언어 의미론, 프로그래밍언어 분석 및 구현, 부분 계산, 함수형 언어 프로그래밍

곽 노 건

1995 서원대학교 전자계산학과, 학사
1996~현재 한양대학교 전자계산학과 석사과정
관심분야 : 프로그래밍언어, 함수형 언어 프로그래밍