Haskell의 펑터와 모나드 프로그래밍

경성대학교 ▌변석우*
부산대학교 ▌우 균*

1. 서 론

1.1 순수 함수형 프로그래밍

함수형 프로그래밍은 람다 계산법, 타입 이론, 의미론 등의 이론적 기반을 바탕으로 오래 동안 연구되어 왔다. 이 기술은 지금까지 주로 대학에서 연구용으로 이용되어 왔으나 최근 이 기술을 실용적으로 이용하려는 시도가 다양하게 진행되고 있다. Ocaml, 하스켈(Haskell), F#, Scala 등의 함수형 언어들이 점차 널리 사용되고 있으며, Java 8, C++-11, C#처럼 기존 언어들도 함수형 언어의 기술을 수용해 가고 있다.

하스켈은 기반 이론들을 충실하게 구현한 순수 함수형(pure-functional) 언어이다 [1]. 이론과 구현 사이의 괴리가 없으므로 이 언어를 통해서 여러 고급 기술들을 명료하고 정확하게 이해할 수 있는 장점이 있다. 하스켈에서는 순수성(purity)을 유지하기 위해서 변수의 값을 동적으로 변화시키는 (mutable variables) 할당문을 사용하지 않으며, side-effect 프로그래밍 기법을 입출력 등의 제한적인 형태로만 이용한다. 순수성에 의하여 하스켈의 함수는 수학적 특징을 유지한다. 함수는 어떤 입력 값에 대해서 결과 값을 갖지 못하거나 (계산이 종료되지 않는 경우), 결과 값을 갖는 경우 그 값은 유일하다. 또한, 결과 값은 오직 입력에 의해서만 결정되어야 하며, 문맥이나 환경 등의 어떤 외부의 요소에 의해서 영향을 받지 않아야 한다. 따라서 한 수식을 같은 값을 갖는 다른 수식으로 치환(substitution)하더라도 등식이 성립된다. 예를 들어, `f(2) = 5` 이면, 임의의 함수 `g`에 대해서 `g(f(2)) = g(5)` 가 성립한다.

1.2 비순수형(Impure) 프로그래밍

프로그램을 순수한 형태로만 코딩하기는 어려우며, 분야에 따라 명령형 프로그래밍처럼 side-effect 방식으로 프로그래밍하는 것이 자연스러울 수 있다. 입출력이 그 대표적인 경우이다. 하스켈에서는 모나드(monad) 프로그래밍을 이용함으로써 한 프로그램 내에서 순수 함수형과 명령형 형태의 프로그래밍을 함께 사용할 수 있다. 입출력을 위한 `IO` 모나드에서는 함수의 순수성이 유지되지 않으므로, `IO` 타입을 갖는 함수는 순수 함수와 구분하여 액션(action)이라고 부른다. 함수와 액션을 동시에 이용할 때는 순수성이 훼손되지 않도록 하는 것이 중요하다. 하스켈에서는 타입을 이용하여 함수가 액션을 호출하지 못하도록 분리시킴으로써 순수성을 유지한다. 이처럼 하스켈은 함수형 프로래밍와 명령형 프로그래밍의 기능을 동시에 지니고 있다 [2].

입출력뿐만 아니라 `Maybe, List, Reader, Writer, State` 등의 타입에 따라 다양한 모나드가 정의될 수 있다 [3]. 이들은 모나드가 아닌 순수 함수형 형태로 정의될 수도 있지만, 모나드에 의한 추상화는 코드를 훨씬 더 간결하게 표현할 수 있는 장점을 갖는다.

본 고에서는 모나드 프로그래밍의 기본 원리에 대해서 설명하고자 한다. 모나드 프로그래밍에는 람다 계산법에 의한 고차함수(higher-order functions), 파라메터라이즈(parameterized) 타입 등과 더불어 카테고리 이론(category theory)이 이용되므로, 모나드를 충분히 이해하기 위해서는 많은 노력이 요구된다. 본 고에서는 함수형 프로그래밍에 대한 기본적인 지식을 가진 독자들을 대상으로 모나드 프로그래밍의 핵심 개념을 설명하고자 한다. 하스켈 프로그래밍의 기본을 위해서 [4, 5] 등을 참고하기 바란다. 커리화된 함수 (Curried functions), 람다 식 (lambda expression), 대수적 데이터 타입 (Algebriac Data Type) 등에 대한 기본적 지식을 전제로, 모나드에 대한 경험이 없는 초

* 종신회원
† 본 연구는 미래창조과학부의 SW컴퓨팅산업원천기술개발 사업의 일환으로 수행하였음(B0101-17-0644, 매니코어 기반 초고성능 스케일러블 OS 기초연구).

보자들을 위해 하스켈 코드를 이용하여 이론보다는 프로그래밍의 관점에서 설명하도록 한다.

본 고의 구조는 다음과 같다. 2절에서는 파라메터라이즈드 타입의 구조에 대해서 설명하고, 3절에서는 펑터를 구성 방법 및 펑터가 갖는 이론적 특징에 대해서 설명한다. 4절에서는 모나드의 핵심인 바인드(bind) 오퍼레이터와 이것이 do 형태의 치장된(sugared) 구문으로 표현되는 모습을 보여준다. 5절에서 결론 및 향후 전망을 기술한다.

2. 타입과 함수의 사상

2.1 함수 형태로 정의되는 타입

타입은 값들(values)을 모음으로서, 하스켈에서는 타입 또한 인수를 갖는 함수로서 정의된다 (이를 parameterized type이라고 함). 예외처리의 한 형태인 계산의 성공/실패를 여부를 표현하는 Maybe 타입의 정의를 고려해 보자.

```
data Maybe a  = Just a | Nothing
```

산술식의 나누기 3/0, 빈 리스트에서 원소를 추출하는 수식 head [] 등처럼 계산을 수행할 수 없는 모든 수식의 값은 Nothing으로 표현된다. Nothing이 아닌 수식은 Just를 붙여 계산이 성공됨을 표현한다. 예를 들어, Just 3, Just (2/3), Just (head [1,2,3]), Just (Just 2) 등의 표현을 사용한다.

타입 정의에는 키워드 data를 사용하며, 타입은 함수의 형태로 표현된다. 위의 예에서 타입 Maybe는 하나의 인수 타입 변수(type variable) a를 갖는데, 이때 a는 Integer, Bool 등의 타입으로 실례화(instanciation) 된다 (a는 3, True 등의 수식이 될 수 없다). 타입의 정의는 하나가 아닌 여러 개의 타입 수식으로 표현될 수 있는데 이들은 or의 의미를 갖는 | 기호로서 구분하여 표현된다. Maybe에 대한 정의는 Just a와 Nothing의 두 가지 경우로 정의된다. 타입 수식은 구성자(constructor)로 시작되는데, 구성자는 함수로서 그 뒤에 $n \geq 0$ 개의 인수를 갖는다. Just는 하나의 인수를 가지며, Nothing은 인수가 없다. Maybe처럼 타입 수식의 왼쪽에 나오는 구성자를 타입 구성자(type constructor), Just나 Nothing처럼 오른쪽에 나타나는 구성자를 데이터 구성자(data constructor)라고 구분한다. 보통 데이터 구성자를 간략히 구성자라고 부른다.

타입 구성자는 타입을 인수로 받아 타입을 구성하며, 데이터 구성자는 수식을 인수로 받아 수식을 구성한다. 예를 들어, Maybe Bool, Maybe Integer 등은 타입이며, Just True, Just (1+2) 등은 수식이다. Just는 하나의 인수를 갖는 함수로서 이 타입은 Just :: a -> Maybe a 로서 표현되며, Just False :: Maybe Bool 이 된다. 독자들은 이 상황을 Ghci 쉘을 이용하여 :t Just, :t Just Bool 등을 확인해 보길 바란다.

2.2 래핑(Wrapping)과 패턴매칭(Pattern Matching)

Maybe처럼 인수를 가지면서 정의되는 타입을 컨테이터(container)라고 부르며, 이는 Java Generics 등의 타 언어에서도 이용되고 있다. 구성자 Just :: a -> Maybe a는 함수로서 어떤 한 수식의 값을 Maybe 컨테이너로 사상(mapping) 한다. 예를 들어, 식 (1+2) :: Integer, ('d'==) :: Char->Bool에 대해서 Just(1+2) :: Maybe Integer, Just(Just(1+2)) :: Maybe(Maybe Integer), Just('d'==) :: Maybe(Char->Bool)가 된다. 이와 같은 사상을 Maybe로 랩한다(wrap), 혹은 Maybe로 리프트한다(lift)고도 말한다.

래핑은 컨테이너 타입의 값을 구성(construction)하는 과정이며, 반대로 컨테이너로서 구성된 값을 해체(deconstruction)하는 역(inverse) 과정이 필요한데 이것은 패턴 매칭이다. 다음은 Maybe, List, BinTree로 래핑된 정수 값을 1만큼 증가시키는 예로서, incM (Just 3) = Just 4이고, incBT tree1 = Node (Leaf 2) (Node (Leaf 3) (Leaf 4), incL [1,2,3] = [2,3,4]가 된다.

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

inc :: Integer -> Integer
inc x = x + 1

incM :: Maybe Integer -> Maybe Inteteger
incM (Just x) = Just (inc x)
incM Nothing  = Nothing

incBT :: BinTree Integer -> BinTree Inteteger
incBT (Leaf x)     = Leaf (inc x)
incBT (Node t1 t2) = Node (incBT t1) (incBT t2)

incL :: [Integer] -> [Integer]
incL []     = []
incL (x:xs) = (inc x) : incL xs

tree1 = Node (Leaf 1) (Node (Leaf 2) (Leaf 3)
```

incM (Just 3)의 계산은 다음의 과정을 거치게 된다.

(1) incM (Just x)의 패턴매칭에서 변수 x의 값이 3으로 바인드된다. 즉, 패턴 칭에 의해 Maybe 컨테이터 안에 있는 Integer 값을 추출한다.

(2) 실제 연산은 x의 값 3에 inc 함수를 적용함으로

써 이루어진다. 이때 적용되는 함수는 리프팅되기 전의 함수로서, (inc 3) :: Integer 이다.

(3) 연산이 적용된 수식을 Just로 래핑함으로써 Just (inc 3) :: Maybe Integer가 된다.

이와 같이, incM 계산은 패턴매칭-연산-래핑의 과정으로서 구성되는데, 이 과정은 incBT, incL에서도 마찬가지임을 볼 수 있으며 여러 곳에서 빈번하게 나타된다.

위의 BinTree 타입 정의는 오른쪽 부분에 BinTree가 나타나는 재귀적 방법이 적용되고 있으며, List 타입 정의도 재귀적이다.

```
data List a = Null | Cons a (List a)
```

List 타입에는 구문적으로 치장(syntactic sugaring)이 사용되고 있다. List 대신 [] 으로 표현하고, List Bool을 [Bool]로 표현한다. 기호 [] 는 타입뿐만아니라 구성자로서도 사용되는데, Null 대신 [], Cons 대신 (:) 으로 표현한다. 수식 Cons 1 (Cons 2 Null) 는 [1,2] = 1:2:[] 에 해당된다.

재귀적 타입의 데이터를 처리하는 함수 또한 incBT와 incL에서 보듯이 재귀적인 형태로 정의된다. 타입과 함수의 재귀성은 위에서 논의된 패턴매칭-연산-래핑의 계산 과정에 영향을 주지는 않으며, 각 타입에 따라 패턴 매칭과 래핑의 과정에만 영향을 주게 된다. Node는 두 개의 BinTree를 묶어 새로운 BinTree를 구성하며, Cons는 새로운 값 a를 리스트의 맨 앞부분에 붙여 새로운 리스트를 구성하고 있다.

3. 펑터(Functors)

3.1 펑터의 구성

함수 odd :: Integer -> Bool은 입력되는 수가 홀수인지를 판가름한다. 이 함수를 이용하여 Maybe 컨테이너 안의 수가 홀수인지를 판단하는 함수는 아래와 같이 정의될 수 있으며, oddM (Just 3) = Just True, oddM Nothing = Nothing 의 결과를 갖는다.

```
oddM :: Maybe Integer -> Maybe Bool
oddM (Just x) = Just (odd x)
oddM Nothing  = Nothing
```

incM과 oddM의 코드는 실제 연산이 이루어지는 (inc x)과 (odd x) 부분을 제외하면 동일하므로 이 둘을 함께 정의하면 코드의 중복이 발생한다. 이 중복의 문제는 고차함수를 적용하여 해결할 수 있다. 패턴매칭-연산-래핑 단계에서 연산에 해당되는 함수를 분리하고, 이를 아래 mapM의 인수 f로 표현하는 기법을 적용한다.

```
mapM :: (a->b) -> Maybe a -> Maybe b
mapM f (Just x)  = Just (f x)
mapM f Nothing   = Nothing
```

incM = mapM (+1), oddM = mapM odd 의 등식이 성립하므로, mapM을 정의하면 다른 두 함수를 따로 정의할 필요가 없다. mapM의 f는 타입이 일치하는 Maybe 컨테이너 내의 모든 데이터를 처리하는 일반적(generic) 기능을 갖는다.

이와 같이 패턴매칭-연산-래핑의 절차로부터 연산을 분리하고, 고차함수를 사용하는 기법은 다른 타입의 경우에도 적용될 수 있다. 리스트 컨테이너 내의 원소 각각에 어떤 연산을 적용하는 map도 이 경우에 속한다. 앞서 정의된 incL 함수는 map (+1)로서 대신할 수 있다.

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = x : map f xs
```

BinTree 컨테이너에 대해서도 다음과 같은 mapBT를 정의할 수 있으며, incBT = map (+1) 이 성립한다.

```
mapBT :: (a -> b) -> BinTree a -> BinTree b
mapBT f (Leaf x)       = Leaf (f x)
mapBT f (Node t1 t2)   = Node (mapBT f t1)
    (mapBT f t2)
```

패턴매칭-연산-래핑의 절차로부터 연산을 분리하고, 고차함수를 사용하는 기법은 모든 컨테이너 타입에 적용될 수 있다. 위의 각 타입에 따라 정의되는 mapM, map, mapBT 함수는 모두 다르지만 overloaded 함수를 이용하면 이들을 모두를 한 함수 이름(예를 들어, fmap)으로 표현할 수 있다. 하스켈에서는 타입 클래스 (type class)가 이 기능을 한다. fmap은 Functor 클래스를 이용하여 정의되고 있다. 다음은 이 클래스의 타입 f에 대해서 [ ], BinTree, Maybe의 instance가 정의된 예이다. 새로운 파라메터라이즈드 타입이 정의된다면 이에 대한 Functor instance에서 그 타입에 적합한 fmap을 정의할 수 있다.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor [] where
   fmap = map

instance Functor BinTree where
  fmap = mapBT

instance Functor Maybe where
  fmap = mapM
```

어떤 한 타입에 대하여 Functor 클래스에 대한 instance를 정의하면, 그에 대한 패턴매칭-연산-래핑에 의한 계산은 fmap으로 표현된다. 이미 하스켈 시스템은 List, Maybe 등의 타입에 대해서는 instance를 정의하여 제공하고 있으므로, 위의 정의없이 fmap (+3) [1,2,3] = [4,5,6], fmap (+3) (Just 10) = Just 13 등을 계산할 수 있으며, 위의 예처럼 BinTree 타입에 대한 fmap을 정의하면 fmap (+3) Node (Leaf 1) (Node (Leaf 2) (Leaf 3) = Node (Leaf 4) (Node (Leaf 5) (Leaf 6)의 계산 결과를 얻을 수 있다.

3.2 펑터의 수학적 특징

하스켈 프로그래밍에서는 수학의 카테고리 이론(category theory)이 적용된다. 타입은 목적(objects)에, 하스켈의 함수들은 사상(morphisms)에 해당되고, 카테고리에서 요구하는 다음의 두 법칙이 성립된다.

```
id . f = f . id = f          (Identity 법칙)
(f . g) . h = f . (g . h)    (함수 합성의 결합법칙)
```

하스켈의 이름을 따서 이를 카테고리 Hask라고 정의한다 [6]. 여기서 (.) 은 두 함수를 합성하는 연산자이고, id :: a -> a, id x = x 인 함수이다. id는 입력과 출력이 동일하므로 한 타입의 원소를 대신 표현하는 기능을 한다.

한 카테고리를 다른 카테고리로 사상할 때 위에 언급한 Identity 법칙과 합성의 결합법칙을 만족하면 이 사상을 펑터라고 부른다. fmap이 (fmap g) 형태의 함수는 펑터로서 다음의 특징을 갖는다. 함수 (g :: a -> b)와 컨테이너 타입 f에 대해서 다음이 성립한다.

```
fmap g :: f a -> f b
fmap id      = id
fmap (g . h)  = (fmap g) . (fmap h)
```

함수 (fmap g)의 입력과 출력 타입은 모두 동일한 컨테이너로서, 이는 이 함수가 사상하는 카테고리의 영역이 동일함을 의미한다. 이 타입의 예로서 (Maybe Integer -> Maybe Bool), ([Bool] -> [Integer]), (Maybe (Maybe Integer) -> Maybe (Maybe Bool)) 등이 될 수 있다.

등식 fmap id = id 에서 왼쪽의 id는 래핑되기 전의 원소를, 오른쪽의 id는 래핑된 후의 원소를 의미한다. 예를 들어, 두 수식이 동일함을 위한 테스트 fmap id (Just False) == id (Just False) 의 값은 True 임을 확인할 수 있는데, 이때 왼쪽의 id는 False를 의미하고 (id False = False) 오른쪽의 id는 Just False를 의미한다 (id (Just False) = Just False).

위의 등식들을 간단히 테스트하는 함수를 다음과 같이 정의하면, functorIdTest (Just 3) = True, functorCompTest (Just 3) = True 임을 확인할 수 있다.

```
functorIdTest x     = fmap id x == id x
functorCompTest x  =
      fmap (inc . inc) x == ((fmap inc) .
      (fmap inc)) x
```

펑터는 컨테이너의 구조를 유지하는 (structure-preserving) 특징을 갖는다. 예를 들어, fmap inc Node (Leaf 1) (Node (Leaf 2) (Leaf 3) = Node (Leaf 2) (Node (Leaf 3) (Leaf 4)로서 컨테이너 안의 값을 변환시키지만 트리 구조 그 자체는 원래대로 유지한다.

지금까지 설명한 대로 펑터는 매우 좋은 특징을 갖고 있으며, 이 개념이 하스켈의 Functor 타입 클래스로 구현됨으로써 프로그램을 간결하고 추상적 형태로 표현할 수 있게 한다. 펑터는 모나드의 전제조건의 역할을 하지만 펑터 그 자체로서도 유용한 프로래밍 기법이라고 할 수 있다.

4. 모나드(Monad)

Functor가 정의된 타입에 대해서 두 함수 return과 join을 정의하면 Monad를 구성할 수 있다. (Monad는 그동안 Functor를 기반으로 구축되는 Functor-Monad 체계를 가졌으나, 2015년 3월에 발표된 Ghc 버전 7.10부터는 Functor-Applicative-Monad의 체계로 바뀌었다. 이 체계에서는 return에 해당되는 기능을 Applicative의 pure로 정의하고, Monad에서는 return = pure 가 되도록 하였다.)

4.1 모나드 클래스의 return

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

모나드 연산의 핵심은 바인드 오퍼레이터 (>>=) 로서

```
m >>= g  = join (fmap g m)
```

의 등식이 성립된다. 즉, 바인드는 Functor의 fmap과 join으로 정의될 수 있는데, fmap한 결과를 join 한다. (join은 flatten의 기능을 한다. Scala에서 바인드는 flatmap이라고 부르는데, 이는 두 함수의 합성 (flat . map) 을 연상시키는 이름이다.)

return과 join에 대한 정의는 다음과 같다. 이 두 함수는 컨테이터 타입 m 에 대하여 return :: a -> m a, join :: m (m a) -> m a 의 타입을 갖는다. return은 어떤 한 값을 컨테이너로 래핑하는 것으로서 구성자를 이용하여 비교적 쉽게 정의된다. 예를 들어, Maybe의 경우 이 타입은 a -> Maybe a 로서 return = Just 가 되며, 리스트의 경우 return = (:[]) 이 된다. 즉, return True = [True] 가 성립된다.

4.2 Join에 의한 Flattening

join은 모나드에서 중요한 역할을 한다. 타입 m (m a) 는 값 a가 컨테이너 m에 두 번 래핑된 경우이다. 예를 들어, Just(Just True) :: Maybe (Maybe Bool), [[False, True], [True], []] :: [[Bool]] 등이 있다. m (m a) 가 m a 로 되는 flattening 과정은 m에 대한 연속된 두 오퍼레이션의 적용을 의미한다. 안쪽 m a 에 대한 오퍼레이션을 적용한 후 얻어지는 결과에 대해서 다시 m 오퍼레이션을 적용하여 m a 의 결과가 되도록 한다. 이때 그 오페레이션은 각 타입 m의 특성에 따라 결정된다.

m이 Maybe인 경우인 Maybe (Maybe a) 형태를 고려해 보자. 이 타입은 성공(Just)과 실패(Nothing)을 점검하는 오퍼레이션을 하는데, 두 연속된 경우는 Just-Just, Just-Nothing, Nothing-Just, Nothing-Nothing의 4가지 경우이다. 그러나 Nothing은 인수를 갖지 않는 상수이므로 마지막 두 경우는 표현될 수 없으며, 오직 첫 번째와 두 번째가 고려될 수 있다. Just-Nothing은 실패한 계산의 결과를 이어서 다른 계산을 수행되는 경우이므로 전체 계산의 결과는 실패(Nothing)로 정의된다. 첫 번째 Just-Just의 경우만이 Just가 될 수 있다. 즉, join (Just Nothing) = Nothing, join (Just (Just x)) = Just x 로 정의된다.

m이 리스트인 경우인 [[a]] 형태를 고려해 보자. 리스트 타입은 임의의 개수 (n≥0)의 원소들의 모음을 표현하는 기능을 한다. 예를 들어, [], [1], [1,1], [1,2,3], [3..] (3부터 시작되는 무한 개의 원소로 구성된 리스트 [3,4,5...]) 등이 모두 [Integer] 타입을 갖는다. [[a]]는 리스트 내에 리스트가 내포된 형태인데, 안쪽 리스트 [a]와 바깥쪽 리스트 [[a]]는 모두 불규칙적인 원소의 수를 갖는다. join은 이러한 특징을 갖는 리스트에 대해서 [[a]] -> [a] 형태의 변환 기능을 한다. 예를 들어, [[1]. [2,3], [], [4]], [[1,2,3,4]], [[], [1], [2,3,4]] 모두 [[Integer]] 타입으로 볼 수 있으며, 이들 각각의 join 결과는 모두 [1,2,3,4] 가 되어야 한다. 이미 하스켈에 정의된 리스트 함수 중에서 concat이 이 기능을 하므로, 리스트인 경우 join = concat 으로 정의된다.

join은 각 컨테이너 타입에 따라 다르게 정의된다. Maybe와 List의 경우를 비교하면, 둘 모두 같은 join 오퍼레이션이지만 이들 각각의 특성은 매우 다름을 알 수 있다. 이런 점이 모나드를 공부하는데 겪는 어려움 중에 하나이다. 한 모나드에 대해서 잘 이해하였지만, 타입 구조가 다른 모나드를 대할 때는 생소하게 느낄 수 있다. 이를 극복하기 위해서는 타입에 대한 이해를 충분히 하는 것이 필요하다.

4.3 바인드 (〉〉=)

바인드의 타입은 (>>=) :: m a -> (a -> m b) -> m b 이다. 여기서 타입 (m a) 는 컨테이너 타입 m 문맥에 따른 계산을 수행한 결과 얻어지는 a 타입의 값을 의미한다. 예를 들어, getChar :: IO Char 인데, 이는 IO 액션 (키보드로부터의 입력) 결과 얻어지는 한 문자를 의미한다. (m a)의 계산 결과 얻어지는 값 a 타입의 값은 함수 (a -> m b) 에게 전달되어 사용된다. 이처럼 바인드는 두 계산 (m a)와 (m b)를 순차적으로 (sequential) 진행시키며, 두 번째 계산 결과 (m b)가 두 연속된 계산의 결과가 된다. 다음은 Maybe 타입에 대한 더하기를 순수 함수형과 모나드 방식의 두 가지로 표현한 예이다.

```
add :: Maybe Int -> Maybe Int -> Maybe Int
add (Just x) (Just y) = Just (x+y)
add  _        _       = Nothing

addM :: Maybe Int -> Maybe Int -> Maybe Int
addM x y =  x >>= \a -> (y >>= \b -> return (a+b))
```

addM (Just 2) (Just 3) = Just 5 의 결과를 얻으며, addM (Just 2) Nothing = Nothing, addM Nothing (Just 3) = Nothing 이 된다. add에 대해서도 동일한 결과를 얻는다. 위의 정의에서 \a -> ... 은 람다 식으로서 수학적으로는 λa. ... 의 표현에 해당된다. 순수 함수형 방식에서는 입력되는 데이터가 Just 인지 아닌지에 대한 점검이 명시적으로 (explicity) 하게 표현하지만, 모나드 방식에서는 이 과정이 숨겨져 (implicit) 있다. x >>= \a 에서 (Maybe Int)를 a로 전달할 때 이 값이 Just 인지의 여부가 내부적으로 이루어지며(이 과정은 펑터에서 처리됨), a에게는 단지 필요한 값만 전달된다. 물론 여기서도 x가 Nothing이라

면, a 다음에 나온는 수식에 상관없이 결과는 Nothing이 된다. 이처럼 모나드로 코딩할 때는 이와 같은 과정은 자동으로 처리되며 실제 연산(이 경우에는 더하기)에 집중하여 표현하므로 가독성이 좋아진다. 위에 정의된 코드는 복잡해 보이는데, 이를 do로 치장된(sugared) 코드로 표현하면 가독성이 좋음을 느낄 수 있다.

```
addM :: Maybe Int -> Maybe Int -> Maybe Int
addM x y =  do a <- x
               b <- y
               return(a+b)
```

Java 8 등에서 Maybe 모나드에 해당되는 Optional 모나드에 대한 논의가 있다. 객체의 값이 Null 인지의 여부를 점검하는 일이 빈번히 일어남으로써 코드의 가독성이 떨어지고, 실수로 이 점검을 하지 않음에 따라 프로그램의 안전성이 떨어질 수 있는데, 모나드가 이 두 문제를 해결할 수 있다.

바인드가 fmap과 join을 이용하여 처리되는 과정은 다음과 같다.

```
addM x y =  x >>= \a -> (y >>= \b -> return (a+b))
   = join (fmap ( \a -> (join (fmap (\b ->
     return(a+b)) y))) x)
```

여기서 fmap이 적용되는 코드가 (fmap (\b -> return(a+b)) y) 의 형태로 되어 있는 점에 유의한다. return(a+b) :: Maybe Integer 의 타입을 가지며, (\b -> return(a+b)) :: Integer -> Maybe Integer의 타입이 된다. 앞서 fmap은 Maybe 펑터에서 fmap :: (a -> b) -> Maybe a -> Maybe b 형태의 타입을 갖는 것으로 설명했지만, 모나드에서는 이 타입이 fmap :: (a -> (Maybe b)) -> Maybe a -> Maybe (Maybe b) 의 타입이 된다. 이 연산의 결과가 다른 곳에 이용되기 위해서는 이 결과 타입이 Maybe b 형태로 전환될 필요가 있으며, 이 과정에서 join 이 적용되게 된다. 예를 들어, 다음의 계산 과정이 발생한다.

```
addM (Just 1) (Just 2)
 = join  (fmap  (\a  ->  (join  (fmap  (\b  ->
   Just(a+b)) (Just 2)))) (Just 1))
 = join  (fmap  (\a  ->  (join  (Just  (Just
   (a+2))))) (Just 1))
 = join (Just (join (Just (Just (1+2)))))
 = join (Just (join (Just (Just 3))))
 = join (Just (Just 3))
 = Just 3
```

4.4 List 모나드

리스트 모나드는 리스트 조건제시법 (List comprehension)과 연관되어 있다.

```
[ [x, x+1] | x <- [10, 20, 30] ]
[ [x, x+1, x+2] | x <- [10, 20, 30] ]
[ [] | x <- [10, 20, 30] ]
```

위의 경우 각각 [[10,11], [20, 21], [30,31]], [[10,11,12], [20,21,22],[30,31,32]], [[], [], []] 이 된다. 이것이 만들어지는 첫 과정은 [Integer] 타입의 한 원소인 x로부터 [10,11], [20, 21], [30,31]이 만들어 진다. 즉 [Integer] -> [Integer] 의 오퍼레이션이 진행된다. 이때 출력 리스트의 원소의 수가 두 개 인 것은 [x, x+1]으로 표현되었기 때문이며, 두 번째의 경우는 세 개이고, 세 번째의 경우는 원소가 없는 빈 리스트이다. 이들 모두 [Integer]의 타입을 갖는다. 즉, 첫 단계에서는 주어지는 수식에 따라 다양한 길이의 리스트들이 만들어 진다. 두 번째 단계에서는 [[Integer]] -> [Integer]을 수행해야 하는데, 이때 앞서 논의한 join이 적용된다.

4.5 State 모나드

State 모나드는 상태를 변환시켜가면서 계산하기 위해서 사용된다. 이에 대한 타입은 다음과 같이 정의된다.

```
data State s v = State (s -> (v, s))
```

여기서 State 는 두 개의 인수를 갖지만 생략된 괄호를 복원하면 (State s) v 가 된다. 이 타입의 정의는 함수 타입 (s -> (v,s))의 형태로 표현되는데, 왼쪽의 s는 입력 상태를 의미하며, 오른쪽의 (v,s)는 계산 결과 얻어지는 v 타입의 값과 변화된 상태 s를 의미한다. 이 (State s) 타입에 대한 모나드는 다음과 같이 정의될 수 있다.

```
instance Monad (Sate s) where
 return v    = State (\s -> (v, s))
 State m >>= f  = Sate $ \s -> let (a, s') = m s
                                   State m'  = f a
                               in m' s'
```

return은 상태 변환 없이 단지 어떤 값 v를 상태 변환의 형태로 구성한다. 바인드는 두 상태 변환의 계산이 순차적으로 이루어지도록 하는 기능을 한다. 이때 첫 번째 계산의 결과 (a, s')이 두 번째 계산 f 에서

사용될 수 있도록 하며, 두 번째 계산의 결과인 (m' s')이 두 연속된 계산의 결과가 된다.

아래 프로그램은 정수, 변수 및 더하기 형태의 산술식에 대한 타입 및 처리기(evaluator)를 코딩한 에이다. 예를 들어, 산술식 1 + x는 이를 ((+) 1 x) 형태로 바꾼 다음 Aexp 타입으로 코딩하면 Add (N 1) (V "x")으로 표현된다. 여기서 변수가 사용되므로 변수의 값을 저장하고 있는 심볼 테이블 Symtab 이 필요하다. 여기서는 두 가지 형태의 처리기 aEval과 aEvalM을 정의하였다. 전자는 순수 함수형 형태이고, 후자는 모나드 방식으로 정의되었다.

```
 data Aexp = N Int | V String | Add Aexp Aexp
type Symtab = [(Var,Int)]

aEval :: Aexp -> Symtab -> Int
aEval (Add a1 a2) s = (aEval a1 s) + (aEval a2 s)

aEvalM :: Aexp -> State Symtab Int
aEvalM (Add a1 a2)  =  do x <- aEvalM a1
                          y <- aEvalM a2
                          return (x+y)
```

aEval과 aEvalM 모두 Symtab을 이용하여 Add 형태의 Aexp 식을 계산하고 있다. aEval에서는 Symtab s가 aEval 함수의 인수로서 명시적으로 표현되고 있다. aEvalM 결과의 타입은 (State Symtab Int)로서 이는 "Symtab을 상태로 이용하여 State 모나드로서 계산을 하고 그 계산 결과로서 Int 타입의 값을 얻음"을 의미한다. aEvalM에서는 Symtab이 aEvalM이 아닌 하부의 State 모나드의 환경에서 이용되고 있다. Symtab은 모나드에서 다루어지며 aEvalM에서는 이를 고려하지 않고 추상적인 수준에서 더하기와 관련된 것만을 표현하고 있다.

첫 번째 식 (x <- aEvalM a1)은 a1을 모나드로 계산한 (v, s)의 결과에서 v만을 취하여 이것을 변수 x에 바인딩 함을 의미한다. 이때 aEvalM의 타입에 따라 x의 값은 Int 타입이다. 이 원리는 y에 대해서도 같은 방법으로 적용된다. 두 정수 값 x와 y를 (+)를 적용하고 return을 적용하여 (State Symtab Int) 형태의 값으로 만든다.

계산 타입이 (s -> (v,s))인 형태를 갖는 모나드는 여러 곳에서 볼 수 있다. IO 모나드에서 s는 모든 입출력 시스템의 환경을 표현하는 World이며, Parser 모나드의 경우에서는 파싱하려는 입력 스트링에 해당된다. 즉, IO 모나드를 위한 타입은 (World -> (a,World))으로, Parser 모나드를 위한 타입은 (String -> (a,String))의으로 표현된다. 이러한 유형의 모나드와 do 표현을 사용하는 코드는 명령형 프로그래밍 기법을 가능케 하고 있다. 프로그램의 수행이 위에서 아래 방향을 수행되고, 각 수식의 계산은 상태를 이용면서 수행되며, 상태는 동적으로 변환될 수도 있다. IO 모나드에서는 다음과 같이 키보드 및 모니터 환경과 상호 작용하면서 프로그래밍을 할 수 있다. 다음 예는 키보드로부터 한 입력받은 한 글자를 화면에 출력하는 코드로서, 명령형 프로그램 형태로 자연스럽게 표현된다.

```
do x <- getChar
   putChar x
```

5. 결론 및 향후 전망

람다 계산법, 타입 이론, 의미론, 카테고리 이론 등을 배경으로 개발된 하스켈에서는 타입을 함수 형태로 정의할 수 있는 고급 수준의 타입 시스템, 고차함수, 펑터, 모나드, 애로우(arrow) 등의 프로그래밍 기법을 적용한다. 이 기술들은 하스켈 그룹에서 약 20년 이상 사용면서 발전되고 검증된 기술이다. 지난 약 10년 전부터 산업계에서도 이 기술들을 활용하기 시작하였고 [7], 다른 언어에서도 이들을 수용하고 있다. Scala에서는 flatmap을 이용하여 모나드 프로그래밍을 할 수 있다 [8]. 향후 이러한 활동은 더욱 활발해 질 것으로 예상한다.

이와 같이 학문적 연구 결과가 확산되고 실용적으로 사용되는 것은 고무적인 일이다. 우수한 프로그래밍 언어는 소프트웨어의 안전성, 생산성, 가독성 등에 큰 역할을 할 수 있다. 하스켈이 갖는 고급 기술들은 이러한 목적에 도움을 줄 수 있을 것이며, 향후 하스켈 기술에 대한 관심이 있기를 바란다.

참고문헌

[1] Haskell Language, https://www.haskell.org/

[2] Peyton Jones, "Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell," In Engineering Theories of Software Construction, 2002.

[3] Brent Yorgey, "Typeclassopedia", In Monad Reader, issue 13, 2011.

[4] Graham Hutton, Programming in Haskell, Cambridge University Press, 2016.

[5] Miran Lipovaca, Learn You a Haskell for Greate Good!, No Scratch Press, 2011.
[6] Category Hask, https://en.wikibooks.org/wiki/Haskell/Category_theory
[7] Industrial Haskell Group, http://industry.haskell.org/
[8] James Iry, “Monads are Elephants” http://james-iry.blogspot.kr/ 2007/09/monads-are-elephants-part-1.html

약 력

변 석 우

1980 숭실대학교 전자계산학과(공학사)
1982 숭실대학교 전자계산학과(공학석사)
1982~1999 ETRI(책임연구원)
1995 Univ. of East Anglia (영국), Computer Science (Ph.D.)
1999~현재 경성대학교 컴퓨터공학부
관심분야: 함수형 프로그래밍, 정형 증명, 프로그래밍 언어

우 균

1991 한국과학기술원 전산학(학사)
1993 한국과학기술원 전산학(석사)
2000 한국과학기술원 전산학(박사)
2000～2004 동아대학교 컴퓨터공학과 조교수
2004～현재 부산대학교 전자전기컴퓨터공학과 교수
관심분야: 프로그래밍언어 및 컴파일러, 함수형 언어, 그리드컴퓨팅, 소프트웨어 메트릭, 프로그램 분석, 프로그램 시각화