모나드 프로그래밍 응용
Application of Monadic Programming

변 석 우
Sugwoo Byun
컴퓨터과학과
경성대학교
swbyun@kyungsung.ac.kr

요 약

이론적 특성을 유지하면서 side-effect, 입출력, 예외처리, 비결정성 등의 특성을 표현하는 것은 순수 함수형 언어의 오랜 숙제였다. Wadler는 Moggi가 프로그래밍 언어 의미론 연구를 하면서 처음 제시한 모나드 이론을 Haskell에 적용시켰으며, 지난 수년간의 프로그래밍을 통하여 모나드 프로그래밍의 유용함이 다양한 분야에서 입증되고 있다. 이제 모나드는 Haskell에서 side-effect, 입출력, 예외처리, 비결정성 등의 특성을 표현하는 표준 기법으로서 인정받고 있다. 본 고에서는 모나드 프로그래밍의 기본 원리와 이 프로그래밍 기법의 응용 사례를 소개한다.

1. 서 론

Haskell, Miranda™와 같은 순수 함수형 언어는 수학적 특성을 갖는 람다 계산법 (lambda calculus)의 원리를 충실히 유지하면서 구현되었다. 따라서 Church-Rosser, 등식 논리 (equational logic), 지역 투명성 (local transparency) 등의 우수한 특성을 지니고 있다. 그러나 side-effect, 입출력, 비결정성 등을 포함한 알고리즘을 표현하는데 여러 부자연스러운 문제점을 가지고 있었다. 근본적으로 기존 수리 논리학이나 람다 계산법에서 이러한 문제에 대한 해법을 제시한 적이 없으며, 따라서 람다 계산법의 원리를 충실히 구현해 온 순수 함수형 언어에서 이에 대한 확고한 구문적 표현을 정의하기가 어려웠다. 이런 현상은 이러한 특성들을 매우 자연스럽게 표현하고 있는 명령형 프로그래밍과는 좋은 대조를 이루고 있다. 이 문제 때문에 순수 함수형 언어의 사용 범위가 입출력이나 상태제어가 잘 발생하지 않는 소위 기호 처리 (symbolic computation) 등으로 제한되는 경향이 있었다.

1989년 Moggi는 '계산 (computation)'에 대한 의미 (semantics)를 카테고리 이론의 모나드 (Monad)를 이용하여 표현할 수 있음을 보였다 [Mog89]. 여기서 계산이란 '값 (value)'과 대비되는 개념으로서, 값은 계산의 결과이다. 일반적으로 프로그램의 의미를 '값에 대한 함수'라고 보는 경향이 많은데, 이런 방법은 계산에 대해서는 아무런 논의를 할 수 없으므로 프로그램에 대한 의미를 추상적으로만 다루게 된다. 프로그램의 의미를 값보다 계산에 근거하

여 제시할 필요가 있다는 것이 Moggi의 동기였다. Wadler는 Moggi에 의해서 추상적으로 제시된 이론을 프로그래밍으로서 구체적으로 표현할 수 있음을 제시하였다 [Wad90]. 그 후 계속된 모나드에 대한 연구결과 [Wad92][Wad95][PW93][LHJ95], 모나드는 Gofer에서 처음 구현되었으며 후에 Haskell 98의 정식 라이브러리로서 채택되었다.

모나드가 Haskell에 채택된 후 모나드를 이용한 많은 프로그램들이 개발되었으며, 모나드는 다양한 방면으로 이용될 수 있는 매우 유용한 도구로서 인정받고 있다. 특히, Haskell이 입출력이 빈번히 발생하는 멀티미디어 프로그래밍에 매우 적합한 언어라고 주장이 제기되고 있는데 [Hud00], 이것은 함수형 언어의 입출력이 더 이상 문제가 되지 않음을 내포하고 있다. Haskell에서 제공되고 있는 모나드 함수 **do**를 이용하여 입출력, 상태 제어, 예외처리 등의 프로그래밍을 고수준에서 (high-level)에서 자연스럽게 프로그래밍할 수 있게 되었다.

본 고에서는 모나드 프로그래밍의 기본 원리와 이 프로그래밍 기법의 응용 사례를 Haskell 프로그램을 이용하여 소개한다. 카테고리 이론에 대해서는 언급하지 않으며, 여기서 논의되는 프로그램들은 실제로 Haskell 인터프리터인 Hugs에서 수행될 수 있다.

2. 순수 함수형 언어의 계산

일반적으로 함수형 언어와 같은 선언적 언어는 “what 만을 기술하고 how는 기술하지 않는다”고 일컫는다. 여기서 how란 계산 과정을 의미한다. 명령형 언어에서 프로그램의 수행 순서를 위에서부터 아래로 명시적으로 기술하는 것과는 달리, 선언적 언어에서는 수행 순서가 표현하지 않는다. 순수 함수형 언어에서는 Church-Rosser의 특성에 따라 수행 순서에 상관없이 계산되는 결과 값은 유일하다. 즉, 한 구문 내에 두 개 이상의 여러 redex가 존재할 때 어떤 redex를 먼저 선정하여 계산하더라도 그 결과 값은 동일하다. 또한 nomalising reduction strategy에 따라 그 값에 대한 계산 방법이 결정될 수 있으므로, 프로그래머가 이 과정을 기술하지 않더라도 프로그래밍 시스템에서 이것을 자동적으로 처리하고 있다.

이러한 원리에 따라 함수형 프로그램은 간결하게 표현될 수 있으나, 문제는 경우에 따라서 수행 순서를 명시적으로 제어하는 것이 필요하다는데 있다. 그 대표적인 예로서 다음과 같은 입출력을 고려한다.

```
f = (display 'a', display 'b')
```

여기서 display 함수는 글자를 화면의 현재 커저가 있는 곳에 연이어 출력하는 함수라고 가정하자. 이때, display 'a'와 display 'b' 두 개의 redex 중에서 전자를 먼저 계산하면 "ab"로 출력될 것이며, 후자를 먼저 출력하면 “ba"가 되므로 수행순서가 중요한 의미를 갖는다. 이와 같이 입출력이나 side-effect 등의 특성을 갖는 알고리즘을 프로그래밍하기 위해서는 계산과정의 명시적 제어가 필요하다.

3. 모나드의 정의

가. Haskell의 타입 클래스

함수형 언어 타입을 정의할 때 타입 변수가 사용될 수 있다. 예를 들어, Tree 타입을 정의함에 있어서 타입 변수 a가 사용된다.

```
data Tree a = Leaf a
            | Branch (Tree a) (Tree a)
```

타입 변수 a의 실례화 (instanciation)에 따라 Tree의 타입이 결정된다. (Tree Int)는 정수로 구성되는 Tree를 의미하며 (Tree Bool) Boolean 값으로 구성되는 Tree를 의미한다. 이때, Tree는 하나의 타입을 인수로 받아서 타입을 결과 값으로 산출하는 함수로서 볼 수 있다. 이런 관점에서 볼 때 Tree는 타입 구성자 (type constructor) 역할을 하며, 타입 인수 없이 그냥 Tree로서 기술되는 경우 타입은 first-class citizen이 되므로 고차 타입 (higher-order type)의 원리가 적용된다고 말할 수 있다.

Haskell에서는 class를 고차타입 방식으로 정의할 수 있다. 예를 들어, Functor class는

다음과 같이 정의된다.

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

이 Functor 클래스에 대해서 f가 Tree나 [ ] (list) 등으로 실례화됨에 따라 여러 instance들이 정의될 수 있다.

```
instance Functor Tree where
fmap f (Leaf x)          = Leaf (f x)
fmap f (Branch t1 t2) = Branch (fmap f t1)
                               (fmap f t2)

instance Functor [ ] where
  fmap f [ ]      = [ ]
  fmap f (x:xs) = fx : fmap f xs
```

위의 class와 instance들을 이용함으로써 fmap은 Tree와 리스트에 모두 이용할 수 있도록 overload 될 수 있다.

나. Monad Class

모나드는 고차타입 (parameterized types)에 대한 *계산(computation)* 과정을 표현한 것이다. 모나드와 관련하여 Haskell은 Monad와 MonadPlus의 클래스를 제공하고 있다. 이 중에서 Monad는 Prelude에 정의되어 있으며, MonadPlus는 Haskell 라이브러리로서 제공되고 있다. Monad 클래스는 다음과 같은 네 개의 오퍼레이터가 정의되어 있다.

```
class Monad m where
(>>=)     :: m a -> (a -> m b) -> m b
return    :: a -> m a
(>>)      :: m a -> m b -> m b
fail      :: String -> m a
g >> k  = g >>= \_ -> k
fail s    = error s
```

어떤 한 고차타입에 대한 계산은 그 고차타입에 대한 Monad 클래스의 인스턴스를 정의함으로써 이루어진다. Monad 클래스에서 fail과 >>는 >>=를 이용하여 이미 정의되어 있으므로, Monad에 대한 **인스턴스**를 만들 때는 return과 >>=를 정의하면 된다. 위의 >> 정의에서 (_ -> k)는 이름이 정의되지 않은 unary 함수를 의미하는 데, _ 은 인수에 대하여 이름을 정의하지 않는 것을 의미하므로 입력된 값을 전혀 이용하지 않음을 의미한다.

고차 타입 m에 대한 인스턴스가 정의되었을 때 (m a)의 값은 a를 결과 값으로 *계산해 내는 프로그램*이다. (m a) 프로그램을 구성할 때는 m 인스턴스에서 정의된 return과 >>=이 사용된다. return은 (unit라고도 불림) 어떤 인수를 받아서 그것을 결과 값으로 전달하는 *단순한 계산*을 하고 있다. >>= (bind라고 불림) 오퍼레이터는 두 개의 계산을 받아들여 첫 번째 계산의 결과를 두 번째 계산의 인수로 전달시킴으로써 첫 번째 계산과 두 번째 계산을 *순차적*으로 합성시키는 역할을 하고 있다. 예를 들어, (g>>=h)는 g의 액션이 h보다 선행됨을 *명시적*으로 표현하고 있다. 위의 모나드 클래스에 정의된 >> 오퍼레이터는 g를 수행한 결과 값을 k를 수행하는 과정에서 전혀 사용하지 않는 >>=의 특수한 경우이다.

다. 입출력 프로그래밍

>>=를 이용하는 입출력 프로그램의 예를 보기로 하자. getStr :: IO [Char]는 키보드로부터 주어지는 스트링을 읽는 함수라고 가정하자. writeFile은 데이터 스트링을 (두 번째 인수) 파일이름 (첫 번째 인수, [Char] 타입)에 출력하는 함수이다. 그러면 키보드에서 입력된 스트링을 받아서 이것을 testFile이라는 파일에 기록하는 함수 readAndStore는 다음과 같이 정의된다.

```
readAndStore :: IO ()
readAndStore
  = getStr >>=
      \x -> writeFile "testFile" x
```

한 결과 입력된 스트링은 x로 전달되어 writeFile에서 사용된다. 이 두 계산과정은 getStr이 수행된 후 두 번째 계산이 이루어지게 된다. 만약 첫 번째 계산 결과를 두 번째 계산과정으로 전달하지 않는 경우라면 >> 오퍼레이터를 사용할 수 있다.

```
writeFile "testFile" "Hello File System" >>
putStr "Hello World"
```

위의 프로그램은 Hello File System 이라는 스트링을 testFile에 출력한 후 Hello World 라는 스트링을 화면에 출력한다. 이와 같이 모나드 오퍼레이터를 이용하여 계산 과정을 순차적으로 제어하는 것이 가능하다.

Haskell에서는 do라는 구문이 정의되어 있다. 뒤에 논의하겠지만, do는 단지 >>=를 사용하기 편리하도록 구문적으로 간단하게 표현한 것이다. 앞의 두 프로그램은 do를 이용하여 다음과 같이 표현될 수 있다.

```
readAndStore
  = do x <- getStr
       writeFile "testFile" x

do writeFile "testFile" "Hello File System"
   putStr "Hello World"
```

이미 Haskell의 Prelude에 IO 모나드가 정의되어 있으므로 입출력을 위한 모나드는 새로 정의하지 않아도 된다.

라. 모나드를 이용한 추상화

다음과 같은 Maybe 타입을 고려해 보자.

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

Maybe 타입은 Just와 Nothing의 두 constructor 로서 구성되어 있다. 이 타입은 계산 과정에서 발생할 수 있는 에러의 가능성을 대비하여 프로그래밍하는 경우 사용된다. 어떤 한 함수를 계산한 결과 값을 전달할 때 제대로 계산이 된 경우이면 Just를 붙여서 전달하고, 계산이 실패했을 경우에는 Nothing을 전달한다. 이 함수의 결과 값을 이용하는 측에서는 결과 값이 Just의 형태로 되어있는지 혹은 Nothing의 형태로 되어 있는지를 보고 계산이 제대로 이루어졌는지의 여부를 판단할 수 있다.

이 타입을 이용하여 다음과 같은 add 함수를 정의하기로 한다. 일반적으로 흔히 생각하는 두 개의 정수 값을 받아서 더하기하는 함수의 기능을 확장하여, add에서는 두 정수 값이 모두 (Maybe Int)의 타입으로 되어있다고 가정하자. 즉, add 함수는 에러일지도 모르는 두 정수 값을 받아서 이 둘을 더하는 함수이다.

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

이 add 함수에서는 주어진 인수가 Just인지 아닌지의 여부를 테스트한 후 계산을 수행하고 그것을 다시 Just나 Nothing의 형태로 변환하고 있다. 그러나, Maybe 모나드를 정의하면 이런 과정을 기술함이 없이 *추상적*으로 프로그래밍하는 것이 가능하다. Maybe 타입에 대한 모나드는 다음과 같이 정의된다.

```
instance Monad Maybe where
  return a    = Just a
  x >>= f     = case x of
                  Just a -> f a
                  Nothing -> Nothing
```

여기서 >>=는 첫 번째 인수 x가 실패한 경우 (즉 Nothing의 값을 갖는 경우) f를 계산하지 않고 즉시 실패한 값을 갖게 된다. 이 인스턴스는 앞서 정의된 Monad 클래스에서 m이 Maybe 타입으로 실례화된 것으로서, >>=와 return은 다음과 같은 타입을 갖는다.

```
>>= :: Maybe a->(a->Maybe b)->Maybe b
return :: a -> Maybe a
```

위에 정의된 add 함수에 대한 Maybe 모나드 버전 addM은 다음과 같이 정의된다.

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

이 addM 함수는 Maybe 타입의 인수들을 받아서 계산하지만 Just나 Nothing의 경우를 테스트하지 않고 마치 정수 값을 계산하는 것처럼 정의된다. 이 테스트 과정은 미리 정의된

Maybe 모나드 오퍼레이터들에 의해서 처리되고 있으며, addM 함수를 정의할 때는 이 과정을 기술할 필요가 없다. 이와 같이 low-level details를 모나드에 표현함으로써 모나드를 이용하는 프로그램은 더욱 추상적인 수준에서 표현될 수 있다.

4. 모나드 프로그래밍

가. Id 모나드

```
data Id a = ID a
          deriving Show

instance Monad Id where
  return x       = ID x
  ID x >>= f     = f x

addI :: Id Int -> Id Int -> Id Int
addI x y =    x >>= \a ->
              y >>= \b ->
              return (a + b)
```

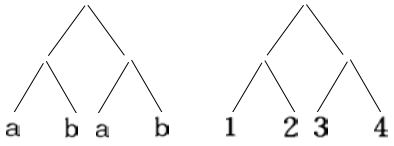
Id 모나드에서 실제로 하는 것은 아무 것도 없으나, Id 모나드를 사용함으로써 계산 순서가 제어될 수 있다. 예를 들어, addI (ID (2+3)) (ID (3+4)) 를 계산하는 데 있어서, 두 redex (2+3)과 (3+4) 중에서 앞의 것을 먼저 계산하게 된다.

나. Label 모나드

이 예는 [Hud00]에서 있다. 다음과 같이 test가 정의되었다고 가정하자.

```
test =    let t = Branch (Leaf 'a') (Leaf 'b')
          in label (Branch t t)
```

test는 아래 그림의 왼쪽과 같은 Tree를 의미한다. 이 Tree의 Leaf의 라벨을 아래의 오른쪽 그림처럼 왼쪽에서 오른쪽으로 순서적으로 정수 값으로 바꾸는 프로그램을 고려하자.



이 프로그램을 작성할 때, 어떤 한 Leaf 노드에 정수 값을 주기 위해서는 바로 전에 사용된 정수 값을 알고 있어야 한다. 명령형 프로그래밍에서는 한 변수를 정의하고 그 값을 동적으로 변환시키는 기법을 이용함으로써 이것을 매우 쉽게 프로그래밍 할 수 있다. 함수형 프로그래밍에서는 side-effect 프로그래밍 대신 튜플 (Integer, Tree Integer)을 이용한다. 튜플의 첫 번째 값은 현재 적용할 정수 값을 의미하며, 두 번째는 지금까지 처리된 Tree의 부분을 의미한다. 이 데이터 구조를 이용하여 다음과 같은 label 함수를 정의할 수 있다.

```
label :: Tree a-> Tree Integer
label t = snd (lab t 0)

lab :: Tree a -> Integer ->
                (Integer, Tree Integer)
lab (Leaf a) n
  = (n + 1, Leaf n)
lab (Branch t1 t2) n
 = let (n1, t1') = lab t1 n
       (n2, t2') = lab t2 n1
   in (n2, Branch t1' t2')
```

위의 프로그램에서 실제로 라벨의 계산은 lab에서 이루어지는데, lab에서는 현재의 정수 값을 lab의 두 번째 인수로 계속 전달시키고 있다. 만약 이것을 명령형 프로그래밍으로 표현한다면 *묵시적인* side-effect 기법으로 이용할 수 있지만, 함수형 프로그래밍에서는 지금의 상태 (state)를 프로그램에서 *명시적*으로 기술해야만 한다. 이 과정은 단조롭고 실수를 유발시킬 수 있으므로, 이 과정을 생략할 수 있는 방법을 모나드 프로그래밍으로서 표현해 보기로 한다. 먼저 Label에 대한 타입과 모나드를 정의한다.

```
newtype Label a
     = Label (Integer -> (Integer, a))
```

Label은 단지 Tree 만을 처리하는 것이 아니라 generic하게 처리할 수 있도록 타입 변수 a를 이용하여 정의되었다. Tree를 처리하기 위해서는 (Label Tree)의 형태로 이용된다.

```
instance Monad Label where
return a    = Label (\s -> (s, a))
Label lt0 >>= flt1
  = Label $ \s0 ->
      let (s1, a1)  = lt0 s0
          Label lt1 = flt1 a1
      in  lt1 s1
```

위의 bind 오퍼레이터 >>=는 두 개의 오퍼레이션 lt0와 flt1을 순서적으로 처리하도록 기술하고 있는데, 이때 lt0는 어떤 주어진 상태 s0를 이용하여 계산하고, 이 결과 산출된 결과값 a1과 상태 s1을 이용하여 그 다음 단계의 계산 lt1을 수행하도록 하고 있다.

앞서 정의된 label과 lab의 Label 모나드를 이용하는 버전은 다음과 같이 각각 mlabel과 mlab으로 정의될 수 있다.

```
mlabel :: Tree a -> Tree Integer
mlabel t =  let Label lt = mlab t
            in snd (lt 0)

mlab :: Tree a -> Label (Tree Integer)
mlab (Leaf a)
    = do n <- getLabel
         return (Leaf n)
mlab (Branch t1 t2)
    = do t1' <- mlab t1
         t2' <- mlab t2
         return (Branch t1' t2')

getLabel :: Label Integer
getLabel = Label (\n -> (n+1, n))
```

위의 프로그램에서 mlab은 lab과는 달리 한 개의 인수만을 가지며 현재의 상태를 인수로서 전달하는 상황이 기술되지 않음을 유의하기 바란다. 상태의 전달은 Label 모나드에서 정의되어 있으며, 이 모나드를 이용하는 측에서는 상태 전달 상황을 기술하지 않고 좀 더 추상적인 수준에서 프로그래밍을 할 수 있다.

이 프로그램 예에서는 모나드를 사용하는 것이 모나드를 사용하지 않는 경우보다 오히려 더 많은 코드를 기술하는 것처럼 보이지만, 프로그램의 규모가 커지면 모나드를 이용함으로써 더 간결하고 명확한 프로그램 의미를 가질 수 있다.

다. State 모나드

이미 앞서 설명한대로 Label 모나드의 핵심은 상태 변환을 정의하는 것이다. Label 모나드는 사실 State 모나드의 한 예이다. State 모나드는 좀 더 일반적으로 다음과 같이 정의될 수 있다.

```
data StateM s a = SM (s -> (s, a))

instance Monad (StateM s) where
  return a = SM (\s -> (s, a))
  SM sm0 >>= fsm1
    = SM $ \s0 ->
        let (s1, a1)  = sm0 s0
            SM sm1 = fsm1 a1
        in sm1 s1
```

독자들은 이미 명령형 프로그래밍을 통하여 상태 변환 기법을 이용한 많은 프로그래밍 예를 경험하였을 것이다. 상태 변환을 이용하는 여러 프로그래밍 예를 State 모나드를 이용하여 프로그래밍 해 볼 것을 권고한다.

라. List 모나드

[ ]는 리스트의 타입 구성자이다. 즉, [Int]이라는 리스트는 실제는 ([ ] Int)을 간략히 표현한 것으로서 Int는 [ ]의 인수로서 볼 수 있다. 리스트 구성자 [ ]에 대한 모나드는 다음과 같이 구성된다.

```
instance Monad [ ] where
 >>=          :: [a] -> (b -> [b]) -> [b]
 return       :: a -> [a]
 m >>= k      = concat (map k m)
 return x     = [x]
 fail x       = [ ]
```

List comprehension은 모나드가 함수형 언어에 도입되기 전부터 사용되어 왔다. Wadler가 Moggi의 모나드 이론을 프로그래밍에 처음 도입하게 된 계기는 list comprehension이 모나드와 밀접한 관계를 가지고 있음을 발견하면서 시작되었다 [Wad90].

List 모나드에 있어서 >>= 오퍼레이터는

list comprehension과 동일한 효과를 가질 수 있다. 또한, 앞서 언급한대로 do와 >>= 는 같은 의미를 갖는 오퍼레이터들로서 단지 구문적인 표현만 다를 뿐이다. 따라서 리스트의 경우, 어떤 한 리스트가 처리되는 과정은 do, >>= 및 list comprehension의 세 가지 형태로 표현될 수 있다. 예를 들어, 다음의 ex1, ex2, ex3는 모두 같은 의미를 갖는다.

```
ex1 = do x <- [1, 2, 3]
         y <- [4, 5, 6]
         return (x, y)

ex2 = [(x, y) | x <- [1, 2, 3],
                y <- [4, 5, 6]]

ex3 = [1, 2, 3] >>= \x ->
      [4, 5, 6] >>= \y ->
      return (x, y)
```

일반적으로 함수 f가 주어졌을 때 다음 세 구문은 같은 의미를 갖는다.

```
do x <- xs ; return (f x)
[f x | x <- xs]
xs >>= \x -> return (f x)
```

마. Parsing 모나드

다음과 같은 Context-Free Grammar에 대한 파서를 고려해 보자.

```
G ::= a G b | c
```

이 룰의 의미는 G룰을 파싱할 때 (a G b)와 c에 대하여 둘 중에 하나에 대하여 파싱됨을 의미한다. 즉, |는 두 파싱을 선택할 수 있음을 의미하는 choice 오퍼레이터의 역할을 한다. 이것을 프로그래밍할 때는 backtracking의 기법이 사용되고 있다. 예를 들어, (a G b | c)에서 먼저 (a G b)에 대해서 파싱을 수행하여 이것이 성공하였으면 G룰의 파싱을 성공적으로 마치고, 그렇지 않으면 두 번째 대안 c에 대해서 파싱을 수행하도록 한다.

함수형 프로그래밍에서 backtracking은 어떤 방법으로 프로그래밍될 수 있나? 본 Parser 모나드에서는 이 점에 중점을 두어 논의하기로 한다. Backtracking에 대한 오퍼레이션은 fail과 함께 정의되어야 한다. Haskell에서는 이미 이와 관련된 오퍼레이터들을 Monad 라이브러리 내의 MonadPlus 클래스로서 정의해 놓았다 (MonadPlus를 이용하기 위해서는 Monad 라이브러리를 import해야 한다). Fail은 MonadPlus에서 mzero로 정의되어 있으며, 인스턴스가 만들어짐에 따라 새롭게 정의되므로 overload 된다.

```
class Monad m => MonadPlus m where
mzero :: m a
mplus :: m a -> m a -> m a
```

위의 클래스 정의에서 (Monad m => MonadPlus m)은 어떤 한 타입 m에 대한 MonadPlus를 이용하자 할 때 m에 대한 Monad를 먼저 정의해야 함을 의미한다.

파싱을 위해서는 두 개의 원소로 구성된 (a, s) 형태의 튜플이 이용되고 있다. a는 지금까지 파싱된 스트링을 의미하며, s는 앞으로 파싱되어야 할 스트링을 의미한다. 파싱이 처음 시작할 때 a는 empty ([])의 상태이고 s는 파싱될 프로그램 텍스트이다. 파싱이 진행됨에 따라 a는 점차 증가할 것이며 s는 점차 감소하여, 파싱이 성공적으로 끝나면 맨 처음 시작할 때와는 정반대로 a는 모든 프로그램 텍스트를 담고 있고 s는 empty이다. 만약 파싱 과정이 모두 다 끝났음에도 불구하고 s가 empty가 아니면 그 파싱이 실패하였음을 의미한다.

파싱을 위한 타입 Parser는 다음과 같이, 현재의 파서의 입력되어 있는 상황을 나타내는 변수 s와 파싱될 유형으로 분류된 스트링 a로서 구성되며, 파싱이 실패할 경우를 고려하여 이 둘을 Maybe 값으로 정의한다.

```
newtype Parser s a = P ([s]->Maybe (a,[s]))
```

Parser 모나드는 Maybe 타입을 고려하여 아래와 같이 정의된다. 첫 번째 파싱을 수행하여 그 결과가 실패하였으면 바인드된 두 파싱 또한 실패한 것이며, 만약 첫 번째 파싱이 성공하였으면, 그 결과 값을 가지고 두 번째 파

싱을 수행한다. 여기서 첫 번째 파싱한 결과 값을 그 다음 단계로 계속 전달시키는 것은 후에 파싱이 실패하였을 경우 적용할 backtracking을 고려하였기 때문이다. 즉, 아래 첫 번째 파싱의 결과로 f에게 전달되는 s′는 후에 backtracking이 전달되는 경우 사용될 수 있는데, s′은 아래 MonadPlus의 mplus 오퍼레이터에게 전달되어, 첫 번째 파싱이 실패한 경우 두 번째 파싱을 적용하는 (b s)의 s로서 이용되고 있다.

```
instance Monad (Parser s) where
return x = P (\s -> Just (x, s))
P m >>= P f
 = P $ \s -> case m s of
                Just (x, s′) -> f x s′
                Nothing -> Nothing
```

CFG 룰의 |에 해당하는 기능은 다음의 mplus로서 정의될 수 있다. mpuls는 두 개의 파싱을 받아들여 첫 번째 파싱을 시도해 본다. 이 결과가 성공적이면 첫 번째 파싱 결과를 mplus 파싱의 결과 값으로서 선택하고 종료한다. 만약 첫 번째 파싱이 실패하였을 경우 mplus의 결과는 두 번째 파싱의 결과로서 결정된다.

```
instance MonadPlus (Parser s) where
mzero = P (\s -> Nothing)
P a `mplus` P b
  = P $ \s -> case a s of
                Just (x, s′) -> Just (x, s′)
                Nothing -> b s
```

하나의 글자를 파싱하는 파서 symbol은 다음과 같이 정의될 수 있다. symbol은 단순히 현재 읽어들인 글자가 미리 룰에 있는 글자와 동일한 것인지를 점검하여 동일한 것이라면 두 번째 튜플의 첫 번째 글자를 첫 번째 튜플로 이동시킨다.

```
symbol :: s -> Parser s s
symbol s
  = P $ \xs ->
        case xs of
          [ ] -> Nothing
          (x:xs′) -> if x == s
                       then Just (x, xs′)
                       else Nothing
```

이와 같이 정의된 모나드 오퍼레이터들을 이용하여 다음의 CFG 룰

G ::= a G b | c

을 파싱하는 프로그램 gram은 다음과 같이 정의될 수 있다.

```
gram = symbol ″a″ `cat` gram `cat` symbol ″b″ `mplus` symbol ″c″
```

여기서 `cat`는 두 스트링의 concatenation을 의미한다. 모나드를 이용함으로써 backtracking을 포함한 모든 파싱 룰이 매우 자연스럽게 프로그래밍으로 표현되고 있음을 알 수 있다.

바. 모나드 composition

기존의 두 함 수 g와 f를 합성하는 (g.f)x에 대응하는 (g `composeM` f)x 형태의 모나드 합성 composeM은 다음과 같이 정의된다.

```
composeM :: Monad M => (b -> m c) ->
(a -> m c) -> (a -> m c)
(g `composeM` f) x = f x >>= g
```

5. 모나드 법칙들

모나드가 구문적으로 여러 다양한 형태로 표현된다고 하더라도 이들 모두는 공통적으로 모나드의 기본법칙 (law)을 만족시켜야 한다. 여기의 법칙이란 카테고리 이론적으로 정의된 모나드의 법칙으로서 이들은 다음과 같은 세 가지로서 정의된다.

(1) Left unit
```
return a >>= k    = k a
```
(2) Right unit
```
m >>= return      = m
```
(3) Associativity

```
m >>= (\x -> k x >>= h)
    = (m >>= k) >>= h
```

위의 법칙은 모나드 오퍼레이터를 이용한 Haskell 구문으로서 표현되어 있으므로, do 문, 혹은 리스트 모나드에 대해서는 list comprehension으로서도 표현될 수 있다. 이와 같이 모나드란 두 개의 기본 오퍼레이터 (return, >>=)와 세 개의 법칙으로서 구성되어 있다. 이 조건을 만족시키는 구문이라면 그것은 모나드가 될 수 있는 것이다.

프로그래밍의 관점에서 볼 때, (1)은 값 a로 구성된 단순한 unit 액션을 다음 액션 k로 보내는 것은 a를 그대로 k로 보내는 것과 같음을 의미한다. (2)는 액션 m의 결과를 unit 액션으로서 return하는 것은 unit 액션으로 return하지 않고 곧바로 액션 m을 결과값으로서 return하는 것과 같음을 의미한다. (3)의 associative law는 h에 x가 자유변수 (free variable)로 나타나지 않는 경우 성립한다. (3)의 특별한 경우인 sequence 오퍼레이터 (>>)에 대해서는 다음과 같은 법칙이 성립한다.

```
m1 >> (m2 >> m3) = (m1 >> m2) >> m3
```

이러한 법칙을 do 구문으로 표현하면 다음과 같다.

```
do x <- return a; k x   = k a
do x <- m; return x     = m
do x <- m; y <- k x; h y
      = do y <- (do x <- m; k x); h y
do m1; m2; m3  = do (do m1; m2); m3
```

6. 모나드 컴비네이터

함수형 프로그래밍에서는 컴비네이터 (combinator) 라이브러리 기법을 사용하고 있다. 컴비네이터란 여러 프로그램 부분들 (program fragments)을 합성하여 새로운 프로그램 (혹은 프로그램 부분)을 구성하는 함수이다. 고차 함수 (higher-order function)의 기능 덕택으로 함수형 프로그래밍에서의 프로그램 합성은 다른 프로그래밍 기법에 비하여 훨씬 더 강력한 기능을 갖는다. 잘 알려진 리스트 처리 컴비네이터로서 map과 filer가 있다. 모나드에서 정의되는 return과 바인드 오퍼레이터 (>>=) 또한 컴비네이터로 볼 수 있다. 컴비네이터를 이용함으로써 프로그램의 합성 과정에서 발생하는 프로그램들 간의 인터페이스를 자동적으로 처리할 수 있으므로 프로그램의 합성을 매우 쉽고 간단하게 할 수 있다. 모나드 컴비네이터는 컴비네이터를 이용하는 프로그램의 합성에 중요한 역할을 하고 있다.

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.

우리는 이미 앞서 Maybe 모나드를 이용한 addM 함수가 데이터를 처리하는 과정에서 Just와 Nothing의 경우를 고려하지 않는 예를 보았다. 사실 이 과정은 low-level details로서 Maybe 모나드에 정의되어 있다. Low-level details가 Maybe 모나드에 캡슐화 (encapsulation) 된 형태로 처리된 덕택으로 addM에서는 이 과정을 고려치 않는 추상적 프로그래밍이 가능한 것이다. Label 모나드의 경우에도 마찬가지로, 상태를 전달하는 과정의 low-level details가 Label 모나드에 캡슐화되어 있어서 프로그래머가 이 과정을 기술하지 않아도 자동적으로 처리되고 있다.

프로그램을 합성하는 측면에서 볼 때, 모나드 컴비네이터는 프로그램 부분들의 ADT (abstract data type)에 대한 표준화된 인터페이스의 역할을 하고 있다. Haskell의 class 타입을 이용한 overloading의 기능과 더불어, 프로그램 합성과정에서 발생하는 인터페이스를 바인드와 return의 표준화된 컴비네이터로서 처리함으로써 더욱 추상화되고 편리한 프로그래밍 환경을 제공하고 있다.

한편 모나드는 generic한 함수 정의할 수 있도록 한다. [Hug00]에 정의되어 있는 다음의 liftM2 함수를 고려하자.

```
liftM2 :: Monad m => (a -> b -> c) ->
(m a) -> (m b) -> (m c)
liftM2 op x y
 = x >>= \a ->
   y >>= \b ->
   return (x `op` y)
```

liftM2 함수는 binary 함수와 두 개의 모나드 형태로 제공되는 인수를 받아서 계산하는 역할을 한다. 따라서 모든 모나드 형태의 데이터를 처리하는 binary 함수는 liftM2로서 처리될 수 있다. 예를 들어, Maybe 모나드의 addM, Id 모나드의 addI는 각각

```
addI = liftM2 (+)
addM = liftM2 (+)
```

로서 정의할 수 있으며, Parser 모나드에서 사용되는 cat는 리스트들을 concatenate 시키는 기능을 함으로, liftM2 (++)로서 정의된다.

7. 향후 연구전망 및 결론

Haskell은 프로그래밍 언어의 최신 기술을 구현하는 연구 목적의 프로그래밍 언어이다 [Haskell]. 많은 좋은 기능들이 구현되고 있으며 GHC (Glasgow Haskell Compiler)의 경우는 비교적 좋은 성능을 내고 있다. 특히, 지난 수년 동안 연구되고 있는 모나드 및 class 타입 등은 매우 강력한 프로그래밍 기능을 제공하고 있다. 프로그램을 구체적으로 제어하면서도, 추상적인 수준에서 프로그램을 합성할 수 있도록 하는 컴비네이터 라이브러리는 함수형 언어만이 갖는 강력한 기능이다.

모나드의 유용성은 이미 다양한 분야의 프로그램 개발을 통해서 입증되었다. 그러나, 모나드를 적용할 수 없는 몇 가지 사례가 보고되었으며, 이를 해결하는 방안으로서 모나드의 기능을 확장시키는 arrow가 제시되었다 [Hug00]. 현재 arrow를 Haskell의 구문으로서 채택하는 문제가 논의되고 있다.

감사의 글

이 연구는 경성대학교 멀티미디어 특성화사업 4차년도 연구비에 의하여 연구되었음.

8. 참고문헌

[Haskell] Haskell home page.
http://haskell.org

[Hud00] Paul Huak. The Haskell School of Expression : Learning Functional Programming Through Multimedia. Cambridge University Press. 2000.

[Hug89] John Hughes. Why functional programming matters. Computer Journal, 32(2), 1989.

[Hug00] John Hughes. Generalisng Monads to Arrows. Science of Computer Programming. June (?) 2000.

[HM96] Graham Hutton and Erik Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4. University of Nottingham, 1996.

[LHJ95] Sheng Liang, Pual Hudak, and Mark Jones. Monad transformers and modular interpreters. In Proceedings of POPL'95. January 1995.

[Mog89] Eugin Moggi. Computational lambda-calculus and monads. In IEEE Symposium on Logic in Computer Science, June 1989.

[PW93] Simon L Peyton Jones and Philip Wadler. Imperative functional programming, In Proceedings of POPL'93, January 1993.

[Wad90] Philip Wadler, Comprehending Monads. In Proceedings of the 1990 ACM Conference on Lisp and Functional Programming Languages. 1990.

[Wad92] Philip Wadler, The essence of functional programming. In Proceedings of POPL'92, 1992.

[Wad95] Philip Wadler, Monads for functional programming. In J. Jeuring and E. Meijer, editors, Advanced Functional Programming, LNCS 925, Springer-Verlag. May 1995.