모나드 프로그래밍의 원리와 응용

경성대학교 컴퓨터공학과
변 석우
swbyun@ks.ac.kr

내용

- 함수형 프로그래밍의 특징
- Parameterized Types에 의한 Contexts
- 펑터(Functor) 스타일 프로그래밍
- Join에 의한 Flattening
- 모나드의 구성
- 모나드 예
- 모나드를 이용한 응용 프로그램
  - 상태(State) 모나드를 이용한 언어 실행기(interpreter)
  - 파서(Parser) 모나드

참고 문헌

- Brent Yorgey “Typeclassopedia”
- The Trivial Monad http://blog.sigfpe.com/2007/04/trivial-monad.html
- Graham Hutton “Programming in Haskell”, CUP, 2007년
- James Iry “Monads are Elephants” (Scala 버전의 Monad)
  http://james-iry.blogspot.kr/2007/09/monads-are-elephants-part-1.html
- Peyton Jones, “Tackling the Awkward Squad: monadic input/output …”

함수형 프로그래밍의 특징

고차함수
- 함수 (프로그램)를 데이터처럼 인수 및 복귀 값(value)으로 사용하고 저장할 수 있음
- (람다 계산법) 모든 구문을 함수로 표현 (예: $2 \equiv \lambda f.\lambda x.f(fx)$)
- 코드 재사용을 극대화 함

순수 함수형 (Haskell)
- 모든 수식은 (문맥에 상관없이) 항상 유일한 값을 갖는다
- 할당문 사용 금지 (상태를 사용하지 않음)
- 리덕션(reduction)에 의한 계산 방법 적용
  - "Substituting equals for equals"
- 선언적: 흐름 제어를 표현하지 않음 (수학의 수식처럼)
  - 예) (1+2) * (3+4)

타입
- 모든 수식은 타입을 가짐.
- Polymorphic 타입 및 타입 추론
- Algebraic 타입
  - 타입 정의를 함수처럼 파라메터를 이용하고 재귀적 방법으로 함
  - 타입 구성자 (type constructor)와 데이터 구성자 (data constructor) : 트리 구조의 표현

함수형 프로그램의 구성

John Hughes (1985) “Why Functional Programming Matters”

기본 개념

- 함수의 순수성: 함수(프로그램의 조각들)는 어떤 환경에서든지 동일하게 작동함
- 재귀적 구성: 함수들을 합성 (synthesis)하여 새로운 함수들을 생성함.
- 컴비네이터(combinators): 프로그램들을 합성하는 함수
  예를 들어, `map`, `filter`, `fold`, …. `bind(>>=)`

문제

컨텍스트에서 동작하는 함수들을 어떤 일관된 방법으로 합성할 수 있을까?
순수 함수뿐만 아니라 impure 한 함수들도 함께 합성할 수 있는가?
이 경우 원래의 순수성은 유지될 수 있는가?

모나드 프로그래밍의 등장

- Moggi의 의미론 연구 (1989년)
- 입출력, 예외처리, jump 등의 해석
  - Computational 타입 TA : "계산 T를 수행하여 얻어지는 값 A"
  - 카테고리 이론 (Category Theory)에 의한 해석

- Haskell의 입출력 등의 비 순수적 계산에 적용
  - Wadler의 논문 참조 (1992년~2000년)
- 입출력뿐만 아니라 모나드는 다양한 분야의 추상적 프로그래밍을 가능케 함.

- 초기에는 생소하게 느꼈지만, 현재 여러 언어에 적용되고 있음

타입의 중요성

- 한 함수의 출력 타입과 다른 함수의 입력 타입이 일치할 때만 합성 가능
- 정확한 타입 정의가 문제 해결에 중요한 역할을 함
  - 정의된 타입의 의도 및 배경을 정확히 파악해야 함.
- 항상 타입에 집중할 것.

모나드의 구성

1. 고차함수
2. Parameterized 타입 (`f a`)
   - Haskell에서 특정 타입은 대문자 (예: `Maybe Int`) , 타입 변수는 소문자로 (`f a`)로 표현.

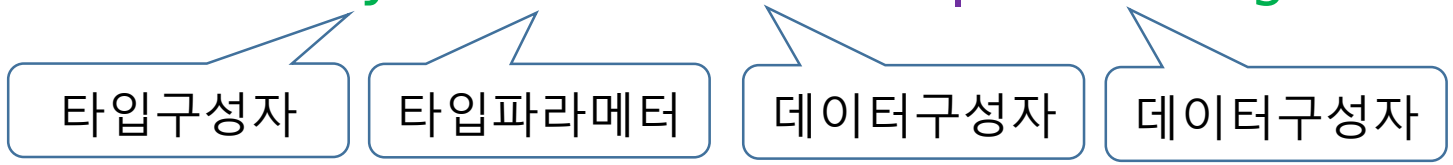
3. (`f, return, join`) 혹은 (`f, >>=`)
   - 펑터(Functor) `f` - 주어진 함수를 주어진 컨텍스트 `f` 안의 값에 적용
     ```
     fmap :: (a -> b) -> f a -> f b
     ```
   - `return` (혹은 `pure`) – 값을 단순히 컨텍스트에 넣음 (아무런 영향을 주지 않음)
     ```
     return :: a -> f a
     ```
   - join – 값이 두 개의 포개진 컨텍스트에 담겨있을 때 이를 하나의 컨텍스트로 만들어 넣음
     ```
     join :: f (f a) -> f a
     ```

   - bind (>>=)
     ```
     (>>=)  :: m a -> (a -> m b) -> m b
     xs >>= g   =  join (fmap g xs)
     ```

Parameterized Types

계산의 실패를 표현하는 타입

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




```
divSafe :: Int -> Int -> Maybe Int
divSafe x 0 = Nothing
divSafe x y = Just (x `div` y)

Just :: a -> Maybe a                Nothing :: Maybe a
```

(Just 3) “정수 값 3을 (Maybe Int)의 값으로 wrapping (혹은 lift)함”
“값을 container 안에 넣음 "

Parameterized Type에 의한 값과 그릇

값 (values)

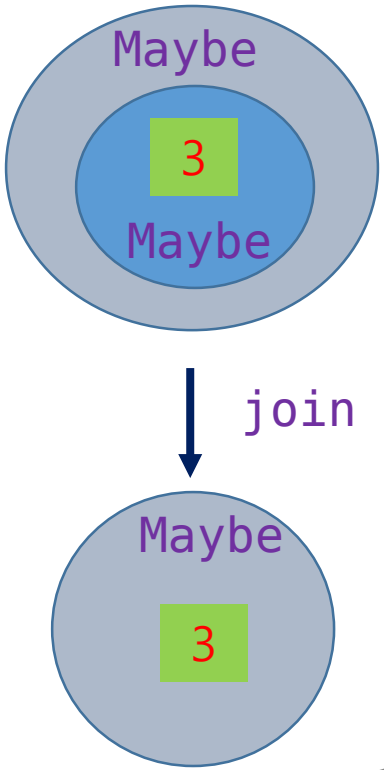
- 그릇에 담기지 않음. 예) `1, 2, 3, True, False, …`

그릇(containers), 환경(Contexts), 혹은 구조 (Structures)라고도 함

- 값을 내포하고 있는 구조체
- 타입 구성자 (type constructor) 를 이용하여 정의되는 타입

그릇의 예

```
  [1,2],  Just 3,  Just(Just 3), Branch(Leaf 3)(Leaf 4),
State(\s -> (3,s))
```



- 주의

  그릇(container)은 직관적 이해를 돕기 위해 사용되는 용어로서 경우에 따라서 이 개념의 적용이 부적절할 수 있음.

  모든 판단은 정의된 코드를 기준으로 이루어져야 함.

다른 예

아무런 효과를 발생시키지 않는 그릇

```
newtype Id a = Id a
```

상태 변환의 각 단계를 (값,상태)의 형태로 표현

```
newtype State s a = State (s -> (a,s))
```

상태가 입출력의 World인 경우

```
type IO a = World -> (a, World)
```

임의의 개수의 값을 담는 수단. List ≡ [ ] (nondeterministic 계산)

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

입력스트링을 파싱하는 경우. 실패의 경우 [ ] 로 표현

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

패턴 매칭, 연산, 그리고 Wrapping

Maybe에 대한 연산의 예

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

Container 타입 계산에 대한 일반적인 현상

1. 입력이 제대로 된 값(즉, Just)인 지를 pattern-matching으로 확인하고 값을 추출함
2. 추출된 값을 원래의 타입의 오퍼레이터로 연산한 다음 (이 경우는 +)
3. 계산 결과를 다시 Wrapping 함 (이 경우는 Just)

유사 코드의 반복됨. 행사 코드(boilerplate code) 를 제거하자!

코딩의 추상화

목적

- 계층화된 구조를 형성하여 코딩을 추상화 시키자.
- 패턴 매칭과 래핑(wrapping)은 하부에 감추고 상부에서는 이 과정을 생략함
- 또한, 이 과정을 고차함수로 정의하고, 필요시 함수 호출함으로써 행사코드를 제거함.

이미 사용중인 map의 예

```
incL :: [Int] -> [Int]
incL []     = []
incL (x:xs) = (x+1) : incL xs

      => map (+1) xs
```

유사한 코딩

```
incM :: Maybe Int -> Maybe Int
incM Nothing  = Nothing
incM (Just x) = Just (x+1)

      => mapM (+1) xs (?)
```

타입 클래스 (Type Class)를 이용하는 overloaded functions 적용

타입 클래스 Functor

map과 mapM의 공통적인 형태

```
fmap ::(a -> b) -> f a -> f b
```

1. 적용함수 (a -> b)
2. 임의의 그릇을 위한 타입 구성자 f와 overloaded 함수 fmap

타입 클래스 (Java의 Interface와 유사)

타입구성자

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

추상메서드

구 현

```
instance Functor [] where
  fmap = map

instance Functor Maybe where
  fmap :: (a->b) -> Maybe a -> Maybe b
  fmap f (Just x)  = Just (f x)
  fmap f Nothing   = Nothing
```

코드 추상화 완성

```
incL :: [Int] -> [Int]
incL []     = []
incL (x:xs) = (x+1) : incL xs

      => fmap (+1) xs
```

```
incM :: Maybe Int -> Maybe Int
incM Nothing  = Nothing
incM (Just x) = Just (x+1)

      => fmap (+1) xs
```

다른 타입에 대해서도 유사한 방법으로 펑터를 정의할 수 있음

펑터는 카테고리 이론으로서, Haskell에서는 그 원리에 따라 구현함.

Haskell을 카테고리 이론적으로 해석 (Category Hask)

- Objects – Haskell의 타입들
- Morphisms – Haskell의 함수들

- 함수의 합성(composition) 오퍼레이터 – (.)
- Identity 함수 - `id`

- Identity 법칙

```
id . f  = f . id  = f
```

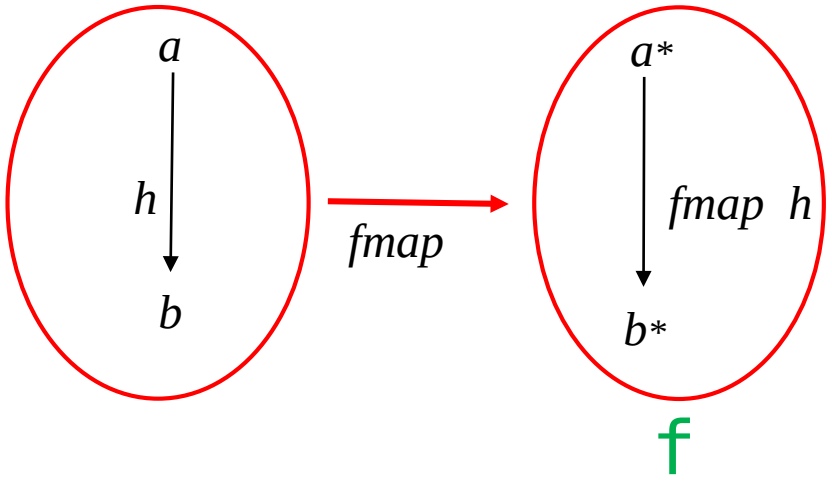
- 결합법칙

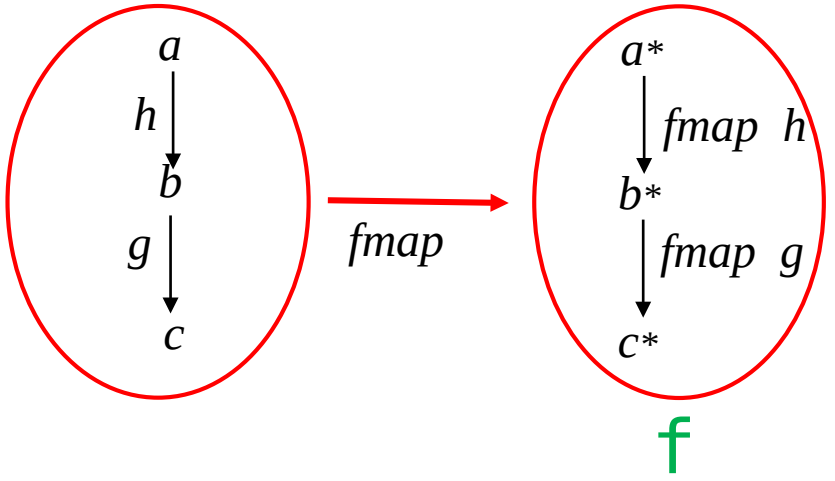
```
(f . g) . h  = f . (g . h)
```

펑터(Functor)의 법칙 (카테고리 이론)

```
fmap :: (a -> b) -> (f a -> f b)
```

```
fmap id_a        = id_a*
fmap (g . h)  = (fmap g) . (fmap h)
```





펑터는 값(타입)뿐만 아니라 함수도 매핑한다.

펑터는 identity와 합성의 구조를 유지한다.

펑터 구현의 예

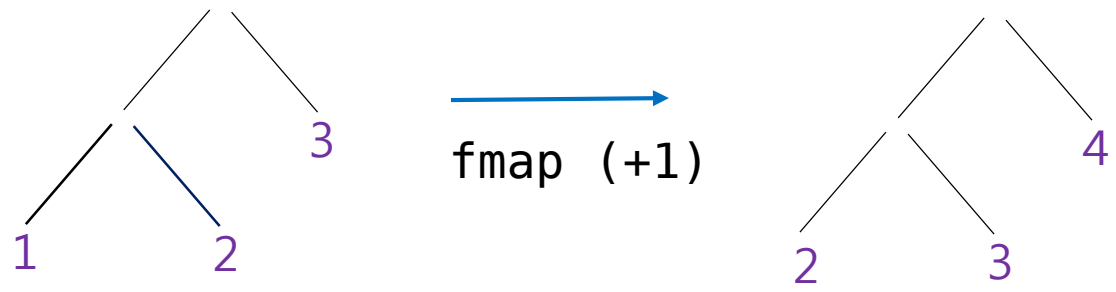
```haskell
newtype Id a = Id a
instance Functor Id where
  fmap f (Id x) = Id (f x)

newtype Maybe a = Just a | Nothing
instance Functor Maybe where
  fmap _ Nothing   = Nothing
  fmap f (Just a)  = Just (f a)

data [] a = [] | a : [a]
instance Functor [] where
  fmap = map
```

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

instance Functor Tree where
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Branch left right)
   = Branch (fmap f left) (fmap f right)
```



Flattening의 필요성

예: 프로그램에 입력되는 데이터 (입력이 없을 수도 있음)

```
parameter :: Maybe String
```

입력된 스트링을 파싱하여 정수형태로 바꿈. 데이터 중 파싱 불가능 자료가 포함될 수도 있음

```
string2int :: String -> Maybe Int
fmap string2int parameter :: Maybe (Maybe Int)
```

연속된 계산이 적용되는 모나드에서 fmap의 적용 함수 타입은 `(a -> f b)`

```
fmap :: (a -> f b) -> f a -> f (f b)
```

f (f b) 타입을 (f b) 형태로 Flattening 할 필요성이 존재함

```
join :: f (f a) -> f a
```

join 정의

- 펑터를 구현하는 fmap은 컨텍스트 내의 값을 변환시키지만, 컨텍스트의 구조를 변화시키지는 않음.
- join은 두 개의 겹치는 컨텍스트를 flattening 하면서 구조를 변경시킴.
- join의 정의는 컨텍스트 타입의 특성을 고려하여 결정해야 함.
- join은 카테고리 이론에서 natural transformation μ 오퍼레이터에 해당하는 것으로서, 모나드 카테고리의 법칙을 만족하도록 정의되어야 함.

join 구현의 예

```
newtype Id a = Id a
joinId :: Id (Id a) -> Id a
joinId (Id as) = as                          -- 단순히 맨 바깥의 구성자를 제거함

newtype Maybe a = Just a | Nothing
joinMaybe :: Maybe (Maybe a) -> Maybe a
joinMaybe (Just (Just x)) = Just x           -- 한번의 Nothing은 전체 Nothing
joinMaybe    _            = Nothing

data [] a = [] | a : [a]
joinList :: [[a]] -> [a]
joinList = concat

newtype State s a = State { runState :: (s -> (a,s)) }
joinState :: (State s (State s a)) -> (State s a)
joinState xss = State (\s -> uncurry runState (runState xss s))
```

모나드의 정의

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

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

바인드 (>>=) 오퍼레이터는 fmap과 join으로서 정의됨

return은 값의 단순한 Lifting. 대부분 데이터 구성자로서 해결

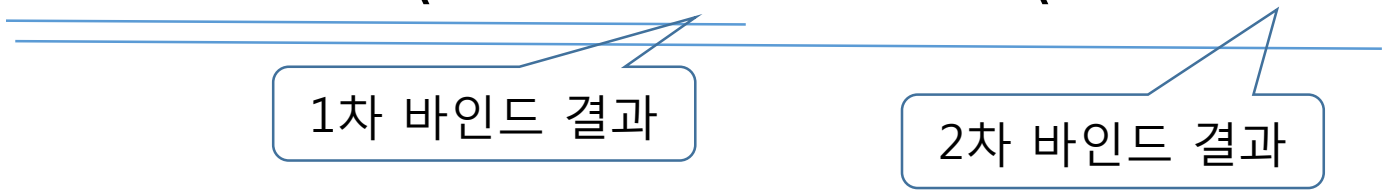
예: `return = Just`, `return = Id`, `return = (:[])`

바인드의 특징과 합성

Bind의 인수 순서를 교환하여 타 함수와 비교

```
fmap :: (a -> b)   -> m a -> m b
fmap :: (a -> m b) -> m a -> m (m b)
bind :: (a -> m b) -> m a -> m b
(<*>) :: m (a -> b) -> m a -> m b      (Applicative)
```

```
m a  >>=  \a -> m b  >>=  \b -> m c
```



모나드의 법칙 (카테고리 이론)

- Left Unit

  return x >>= f = f x

- Right Unit

  m >>= return = m

- 합성에 대한 결합법칙 성립

  m1 >>= (λx.m2 >>= λy.m3) = (m1 >>= (λx.m2)) >>= (λy.m3)

Maybe 모나드의 수행 과정

```
instance Monad Maybe where
  return           = Just
  Just x >>= k     = k x
  Nothing >>= k    = Nothing
```

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

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
```

return 계산이 반드시 마지막에 일어나지 않음

```
fmap (\b -> Just (a+b)) (Just 2)
   = Just $ (\b -> Just (a+b)) 2
   = Just (Just (a+2))
```

타입 (t -> f t)

integer operator

List Monad 예

```
instance Monad [] where
  return x  = [x]
  xs >>= f = concat (map f xs)
```

List Comprehension

```
concat [ [x]++[x+1] | x <- [10,20.30] ]
```

```
[10,20,30]
>>= \x -> [x, x+1]
= [10,11,20,21,30,31]

[10,20,30]
>>= \x -> [x, x+1]
>>= \y -> if y > 20 then [] else [y,y]
= [10,10,11,11,20,20]
```

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


do  x <- [10, 20, 30]
    y <- [x, x+1]
    if y > 20 then [ ] else [y, y]
```

do (>>=의 Sugaring)

```
do { x← e; s }          =    e >>= \x → do { s }

do { e; s }             =    e >> do { s }

do { e }                =    e
```

```
do {  x ← do { y ← m2 ;          =      do {  y ← m2;
                m1 } ;                        x ← m1;
      m3 }                                    m3 }
```

```
= do  y ← m2          (Indentation 정렬에 유의)
      x ← m1
      m3
```

do-Sugared 표현

순수 함수형 표현

```
addMaybe :: Maybe Int → Maybe Int → Maybe Int
addMaybe (Just x) (Just y) = Just (x + y)
addMaybe   _         _     = Nothing
```

모나드의 do-sugared 표현

```
addMaybe :: Maybe Int → Maytbe Int → Maybe Int
addMaybe x y =  do a ← x
                   b ← y
                   return (a+b)
```

State 모나드

- 명령형 프로그래밍의 mutable 변수를 사용하는 경우
  예) 45 + x * (y + z)
  상태에 있는 x, y, z에 해당되는 값을 찾아서 계산한다.
  상태는 할당문 (assignment) 이 포함된 문장을 수행할 때 변화된다.
- 상태는 [(변수, 값)] 으로 표현
- 상태 변환을 나타내는 타입 (Transition 함수로 정의됨)

```
newtype ST s v = ST (s → (v,s))
```

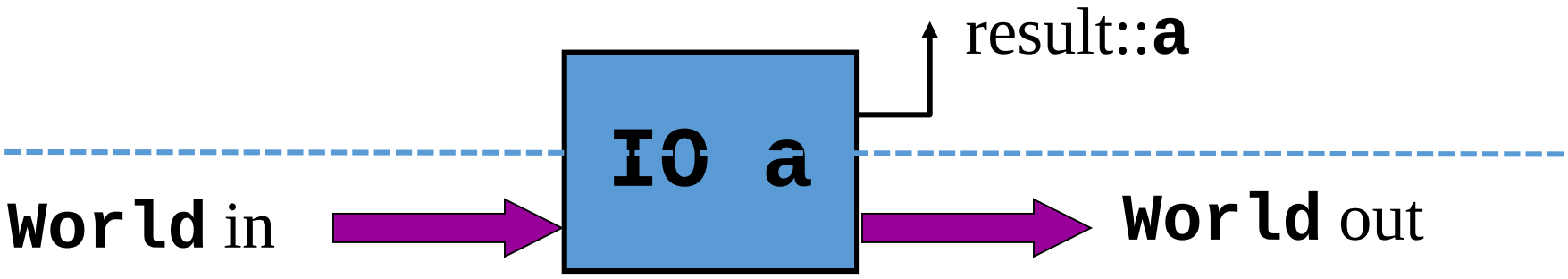
```
instance Monad (ST s) where
 return v    = ST (\s → (v, s))
 ST m >>= f  = ST (\s → let (a, s') = m s
                            ST m'= f a
                        in m' s‘ )
```

m을 계산할 때 현재 주어지는 상태 s를 이용하여 계산함. 이 계산 결과 a와 변화된 상태 s′ 가 중간 결과로서 얻어진다. a를 이용하여 두 번째 evaluation (f a)에 의해 얻어지는 m′ 에 s′을 적용한 것이 최종 결과이다.

IO 모나드

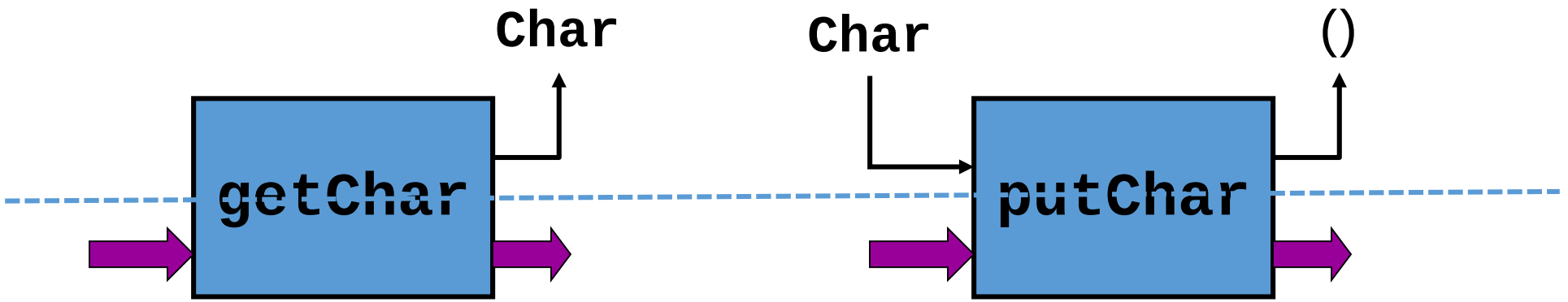
```
type IO a = World -> (a, World)
```

← World의 transition



(>>=)에 의한 두 액션의 합성

(>>=) :: IO a -> (a -> IO b) -> IO b

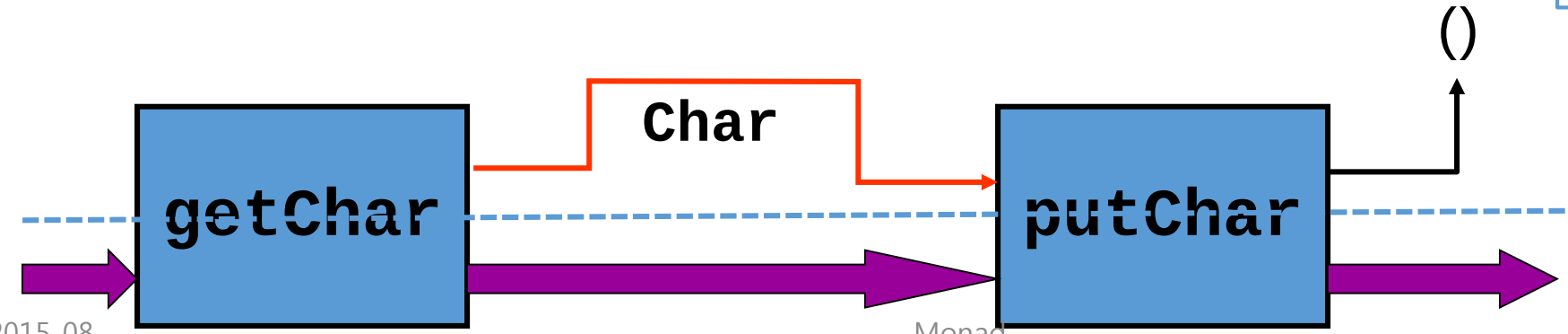


getChar :: IO Char

putChar :: Char → IO ()

getChar >>= \x → putChar x

```
do  x ← getChar
    putChar x
```



IO 액션

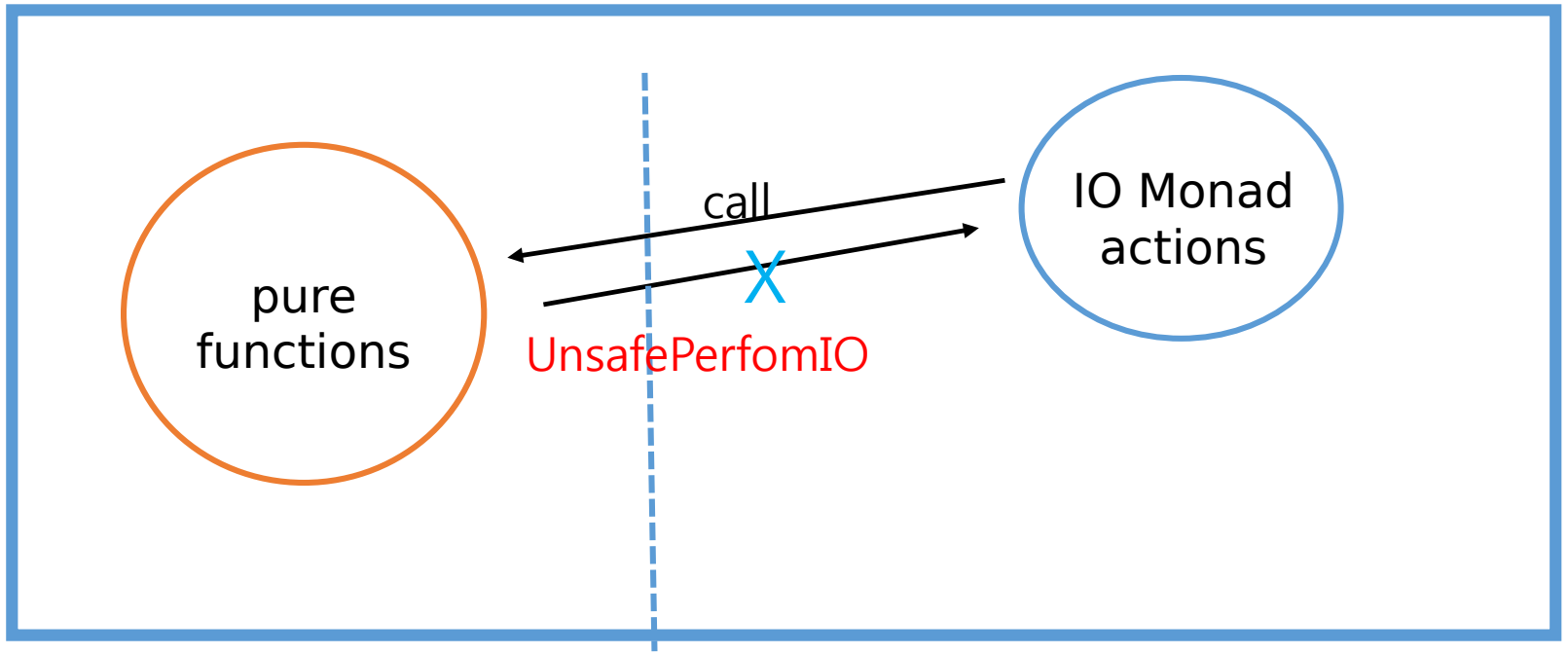
- getChar :: IO Char
  (키보드로부터 한 글자를 읽어 들임)
- putChar :: Char → IO ( )
  (주어진 문자 하나를 화면에 출력, IO 액션을 수행한 결과 얻어지는 값이 없음. ( )로 표현)

IO 모나드의 특징

- IO는 main 함수의 타입으로서, 프로그램에서 반드시 한 번 사용하게 됨.
- 액션 IO a 를 수행한 결과 값 a는 유일하지 않다
- 즉 IO 모나드에서 순수성은 유지되지 않음
- IO 모나드의 비 순수성이 프로그램 전체에 어떤 영향을 미치나?

순수와 비순수의 분리 (Type Checking)

- 한 프로그램 내에서 functions과 actions을 혼합할 때, 이 둘은 어떻게 구분되어 표현되는가? => 타입 및 do 구문
- 이때 순수성은 어떻게 유지될 수 있나? => 제약된 모나드 사용
- 예외적으로 UnsafePerfomIO 함수에서 IO Monad의 액션을 호출할 수 있음



액션이 일등급 시민

```
repeatN :: Int -> IO () -> IO ()
repeatN 0 a = return ()
repeatN n a = do { a; repeatN (n-1) a }
```

```
repeat 10 (putChar ‘x’)
```

- 일등급 시민의 특성은 어떤 특정한 응용 프로그램에 특화된 제어 구조를 가능케 함.
- 제어 구조의 정의 기능과 여러 구문적 Sugaring을 적용하여 Domain Specific Language의 표현을 가능케 함 (Parser 모나드 참조)

전망

- 카테고리 이론의 모나드가 함수형 언어에서 성공적으로 구현됨.
  - 프로그램언어의 학문적 기반이 더욱 강화되고 확대됨.
  - 카테고리 이론 교육의 필요성 제기됨
- 모나드는 함수형 프로그래밍의 보편적 기법이 되고 있음
  - Scala (flapMap과 for), F#, Java 등
- 모나드 외에 중요 프로그래밍 기법
  - Arrow, Applicative, Monoid 등
- Domain Specification Language을 이용한 응용 프로그래밍 확산

응용 프로그램

1. 명령형 언어 실행기
2. Parser 모나드 (Graham Hutton)

명령형 언어의 Interpreter

1. 입력 스트링 (프로그램 코드)에 대한 Scanning및 Parsing을 수행하여 이를 AST (Abstract Syntax Tree)로 표현된 형태
2. AST는 Haskell의 타입으로 자연스럽게 코딩될 수 있음.

```
type Var = String
data Aexp = N Int | V Var | Add Aexp Aexp | Mult Aexp Aexp | Sub Aexp Aexp
data Bexp = T | F | E Aexp Aexp | Le Aexp Aexp | Neg Bexp | And Bexp Bexp
data Stm = Ass Var Aexp | Skip | Comp Stm Stm | If Bexp Stm Stm | While Bexp Stm

type State = [(Var,Int)]
```

```
(코딩 예)  1 + (x + 3)
Add (N 1) (Add (V "x") (N 3))

!((x <= 2) & true)
Neg (And (Le (V "x") (N 2)) T)
```

코딩 및 Parsing 예

- Source Code를 파싱하여 그 결과를 Abstract Syntax Tree 형태로 변환함 (Algebraic 타입으로 표현)

- Factorial 계산 (결과 값은 y에 저장됨)

```
x := 4;
y := 1;
while !(x=1) {
   y := y*x;
   x := x-1
}
```

```
fac4 :: Stm
fac4 = Comp (Ass "x" (N 4))
            (Comp (Ass "y" (N 1))
                  (While (Neg (E (V "x") (N 1)))
                         (Comp (Ass "y" (Mult (V "y") (V "x")))
                               (Ass "x" (Sub (V "x") (N 1)))))))
```

산술식 Interpreter (순수 함수형)

산술식 interpretation의 결과는 값 (values)

```
aEval :: Aexp → State → Int
aEval (N n) s           = n
aEval (V var) s         = let Just n = lookup var s  in n
aEval (Add a1 a2) s     = (aEval a1 s) + (aEval a2 s)
aEval (Mult a1 a2) s    = (aEval a1 s) * (aEval a2 s)
aEval (Sub a1 a2) s     = (aEval a1 s) - (aEval a2 s)
```

State가 명시적으로 parameter로서 표현됨

산술식 Interpreter (State Monad 이용)

```
aEvalM :: Aexp → State → Int
aEvalM aexp initState   = fst $ runST (a_exec aexp) initState      -- 결과 값은 (v,s) 중 v에  저장됨.

a_exec ::  Aexp → ST State Int
a_exec (N n)            =  return n
a_exec (V var)          =  do  s ← ST (\state → (state, state))
                               let x = (let Just n = lookup var s in n)      -- 순수 함수형 계산
                               return x

a_exec (Add a1 a2)    =  do  x ← a_exec a1
                             y ← a_exec a2
                             return (x+y)

a_exec (Mult a1 a2)   =  do  x ← a_exec a1
                             y ← a_exec a2
                             return (x*y)
```

문장에 대한 Interpreter (순수 함수형)

명령형 언어의 Operational Semantics에 따라 State Transition 으로 해석
문장을 Interpret 한 결과는 State
State가 evaluation 할 때 파라메터로서 명시적으로 표현됨

```
evalStm :: Stm → State  → State
evalStm (Ass var aexp) s          = update var (aEval aexp s)  s        -- State가 변화됨.
evalStm (Comp stm1 stm2) s        = evalStm stm2 (evalStm stm1 s)    -- 첫 문장의 수행 결과는 두 번
째의 입력으로 사용됨.
evalStm Skip s                    = s
evalStm (If be stm1 stm2) s       = if (bEval be s) then evalStm stm1 s  else evalStm stm2 s
evalStm (While bexp stm ) s       = if (bEval bexp s)  then evalStm (Comp stm (While bexp stm)) s
                                      else evalStm Skip s
```

문장에 대한 Interpreter (State Monad)

```
evalStmM :: Stm → State → State
evalStmM stm initState  = snd $ runST (exec_stm stm) initState
exec_stm :: Stm → ST State ()
exec_stm (Ass var aexp)        = do  s ← ST (\state → (state, state))    -- 현재의 상태를 읽음
                                     v ← a_exec aexp
                                     let s' = update var v s              -- 상태에서 주어진 (변수, 값) 변경
                                     ST (\ _ → ((), s'))                  --  Update된 상태를 Context로 넣음.

exec_stm (Comp stm1 stm2)  = do exec_stm stm1
                                exec_stm stm2

exec_stm Skip                  = ST (\state → ((), state))
exec_stm (If be stm1 stm2)     = do  b ← b_exec be
                                     if b then exec_stm stm1 else exec_stm stm2

exec_stm (While be stm)        = do  b ← b_exec be
                                     if b then exec_stm (Comp stm (While be stm))  else exec_stm Skip
```

프로그램 수행 결과

- `evalStm fac [("x", 10), ("y", 20), ("z", 30)]`
  `=> [("x", 1), ("y", 24), ("z", 30)]`

Parser Monad 타입 (Graham Hutton의 교재)

- 파서 타입 (State와 유사함)
  ```
  newtype Parser a  =  P (String -> [(a, String)])
  ```
- Context의 State가 입력 String, Parsing된 결과는 a에 해당됨
  파싱이 진행됨에 따라 String(상태)는 앞 부분부터 사라지게 됨.
- a는 상황에 따라 여러 가지 형태로 표현될 수 있음.
  AST 혹은 Int (직접 Interpretation을 할 경우)
- Parsing이 Fail 되는 상황을 표현하기 위해서는
  ```
  newtype Parser a  =  P (String -> Maybe (a, String))
  ```
  Fail을 Nothing 대신 empty list [] 로 표현하기로 함.
- 기본 라이브러리 함수는 Parsing.lhs 파일로 제공 (120라인)

Parser Monad 정의

```
instance Monad Parser where
  return v     =  P (\inp → [(v,inp)])
  p >>= f      =  P (\inp → case parse p inp of                -- 연속된 파싱
                              []      → []                    -- 중간에 한번이라도 실패하면 모두 실패로 간주
                              [(v,out)] → parse (f v) out)    -- 처음 파싱의 결과를 두 번째 파싱에서 이용

instance MonadPlus Parser where
  mzero          =  P (\inp → [])
  p `mplus` q    =  P (\inp → case parse p inp of
                              []      → parse q inp           -- 실패: 두 번째 룰로 파싱
                              [(v,out)] → [(v,out)])          -- 성공: 두 번째 룰의 파싱을 시도하지 않고 종료

(+++)    :: Parser a → Parser a → Parser a          -- Choice
p +++ q     =  p `mplus` q
```

Context Free Grammar에 대한 Parser 정의 (Nondeterminsitic 룰 코딩)

- CFG 룰은 Right-Recursive하고 Factoring 된 형태의 EBNF 룰로 변환

```
expr  ::= term ('+' expr | '-' expr | empty)
term  ::= factor ('*' term | '/' term | empty)
factor ::= '(' expr ')' | number | identifier
```

```
aExp :: Parser Aexp
aExp  =
do t ← term
    do{ symbol "+" ; e ← aExp ; return (Add t e)}
     +++
     do{ symbol "-" ; e ← aExp ; return (Sub t e)}
     +++
     return t
```