JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 28 No. 2, pp. 111-119, February 2023
https://doi.org/10.9708/jksci.2023.28.02.111

Feasibility Study Of Functional Programming In Scala Language By Implementing An Interpreter

Sugwoo Byun*

*Professor, Dept. of Computer Science, Kyungsung University, Busan, Korea

[Abstract]

In this paper, we investigate the feasibility of functional programming in the Scala language. The main issue is to what extent Scala is able to handle major properties of functional programming such as lambda expression, high-order functions, generic types, algebraic data types, and monads. For this purpose, we implement an interpreter of an imperative language. In this implementation, the same functional programming techniques are applied to both Haskell and Scala languages, and then these two versions of implementations are compared and analyzed. The abstract syntax tree of an imperative language is expressed as algebraic data types with generics and enum classes in Scala, and the state transition of imperative languages is implemented by using state monad. Extension and given, new features of Scala, are used as well.

▸ **Key words**: Scala 3, Haskell, State Monad, Interpreter, Algebraic Data Type

[요 약]

본 논문에서는 스칼라 언어의 함수형 프로그래밍 기능의 타당성에 대해서 검토한다. 주된 관심사는 스칼라가 어느 정도까지 람다 식, 고차 함수, 제너릭 타입, 대수적 타입, 모나드 등 함수형 프로그래밍의 주요 특성을 표현할 수 있는 가에 있다. 이 목적을 위하여 명령형 프로그래밍 언어의 인터프리터를 구현한다. 동일한 함수형 프로그래밍 기법을 적용하여 인터프리터를 하스켈과 스칼라로 구현한 다음, 이 두 버전의 구현을 비교 분석한다. 명령형 프로그래밍 언어의 추상 구문 트리는 스칼라의 제너릭스를 갖는 대수적 타입과 enum 클래스로서 표현되고, 명령형 프로그래밍의 상태 변환은 상태 모나드를 이용하여 구현된다. 또한 스칼라의 새로운 기능인 extension과 given도 사용된다.

▸ **주제어**: 스칼라 3, 하스켈, 상태 모나드, 인터프리터, 대수적 데이터 타입

• First Author: Sugwoo Byun, Corresponding Author: Sugwoo Byun
*Sugwoo Byun (swbyun@ks.ac.kr), Dept. of Computer Science, Kyungsung University
• Received: 2023. 01. 30, Revised: 2023. 02. 14, Accepted: 2023. 02. 20.

http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

I. Introduction

스칼라(Scala) 언어는 JVM 상에서 동작하면서 자바 언어와 호환성을 갖도록 설계된 하이브리드 언어이다 [1]. 객체지향 프로그래밍 환경 기반에 함수형 프로그래밍 기법들을 추가함으로써 객체지향과 함수형 프로그래밍 [2]의 두 기능을 모두 사용할 수 있는 환경을 제공한다. 자바 8 (Java 8) [3] 과 코틀린(Kotlin) [4] 또한 스칼라와 유사한 형태로 함수형 프로그래밍을 지원하고 있다.

한편, 가장 대표적인 함수형 언어는 하스켈(Haskell) [5] 이라고 할 수 있다. 본 연구의 동기는 "객체지향 기반 언어인 스칼라, 코틀린, 자바 언어들이 하스켈 만큼의 함수형 프로그래밍 기능을 지원할 수 있는가?" 에 대한 질문에 있다. 스칼라의 함수형 프로그래밍은 수준이 높고 하스켈에 근접해 보이며, 스칼라의 이러한 함수형 프로그래밍 기술은 자바 및 다른 JVM 언어 발전에 영향을 줄 것이다.

본 연구에서는 하스켈의 주요 기능들을 스칼라로 코딩해 봄으로써 스칼라의 함수형 프로그래밍 기능을 점검한다. 이를 위해, 앞선 연구에서 하스켈로 구현된 명령형 프로그래밍언어의 인터프리터 [6] 를 최신 버전에 맞게 수정하고, 이 프로그래밍의 것과 동일한 방법을 적용하여 스칼라 버전의 인터프리터를 구현한 다음, 이 둘을 비교 분석한다. 명령형 프로그래밍 언어의 추상 구문 트리는 스칼라에서 대수적 데이터 타입 (algebraic data type)과 enum 클래스로서 표현되고, 명령형 프로그래밍의 상태 변환은 상태 모나드 (state monad) 를 이용하여 구현된다. 이 과정에서 람다 계산법 (lambda calculus), trait, 제너릭스 (generics), case 클래스에 의한 패턴 매칭, 스칼라 3의 새로운 기능인 extension과 given을 적용한다.

II. Backgrounds

1. Haskell: implementation of mathematics

하스켈은 수리 논리학적 이론을 충실히 구현한 함수형 언어이다. mutable 변수를 사용하지 않으며, 선언적 (declarative)이고, 고급 수준의 타입을 사용하며, 부작용 (side-effect) 기능을 사용하지 않는 순수 (pure) 함수 위주로 프로그래밍하도록 되어 있다. 상태(state)를 이용하지 않으며, 문장(statements) 없이 수식(expression) 만으로 함수를 구성한다. 수학적 특성이 잘 반영된 언어이다.

그러나 실제 프로그래밍의 모든 상황을 수학적 원리를 지키면서 코딩하는 것은 비효율적이므로, 순수/비순수 함수 프로그래밍을 적절히 혼합하는 것이 필수적이다. 예를 들어, 입출력은 상태를 사용하고 부작용 함수로서 표현하는 것이 훨씬 편리하고 자연스럽다.

하스켈에서 모나드를 도입하게 된 주목적은 입출력을 명령형 프로그래밍 형태로 표현하는 것이었다 [7]. 처음에는 낯설게 느껴졌지만, 모나드의 유용성은 갈수록 인정받고 있다. 하스켈의 bind (>>=) 함수와 스칼라, 코틀린, 자바 등에서 사용되는 flatMap이 대표적인 모나드 함수이다.

2. Scala: Hybrid of OOP and Functional

스칼라는 처음부터 자바와 호환성을 가지면서 객체지향과 함수형 프로그래밍의 기능을 동시에 갖는 언어를 목표로 설계되었다. 이 언어의 설계자 Martin Odersky는 하스켈 그룹의 학자들과 공동 연구를 진행하면서 협력해 온 것으로 보인다. 두 프로그래밍을 통합하기 위해서는, 객체지향 프로그램을 함수형 프로그래밍 관점으로 이해하고, 또한 역 방향에 대한 이해를 적용하는 것이 필요하다. 클래스는 타입으로, 클래스 간의 상속은 서브 타이핑으로, 메소드는 함수로서 동작할 수 있는 환경이 구축되어야 한다. 역으로 람다 식에 의한 함수 또한 클래스에 소속된 메소드로서 동작할 수 있어야 한다.

본 연구에서는 하스켈 프로그래밍, 스칼라 프로그래밍, 컴파일러의 파싱 등의 기본 지식을 전제로 설명하고 있다 (모나드 파서는 [8] 참조).

III. Implementation

1. ADT for AST

하스켈에서 대수적 데이터 타입(Algebraic Data Type, ADT)은 합(Sum)과 곱(Product) 타입으로 구성되어 있다. 합은 논리적으로 Or의 의미로, 곱은 And의 의미로 해석될 수 있다. 곱은 튜플(tuples) 혹은 레코드(records)로서 고정된 수의 필드 (field)로 구성된다.

Fig, 1은 산술식, 논리식, 문장들에 대한 AST를 표현하는 대수적 타입을 하스켈로 코딩한 것이며, Fig. 2은 그에 해당되는 스칼라 코드이다. Aexp는 다섯 개 타입들의 합으로 구성되어 있으며, N과 V는 1개의 필드로, Add, Mult, Sub는 2개의 필드로 구성된 곱 타입이다. 예를 들어, 산술식에 대한 스트링 "1 + (x + 3)"을 파싱한 추상 구문 트리는 Add (N 1) (Add (V "x") (N 3)) 으로 표현된다.

스칼라에서는 대수적 데이터 타입을 enum과 trait로서 표현할 수도 있는데, 여기서는 간결함을 위해서 enum과 case

class로서 표현하였다. 하스켈에서는 Aexp (arithmetic 수식) 과 Bexp (논리식)으로 분리하여 표현하였으나, 스칼라에서는 Aexp와 Bexp 없이 이 둘을 하나로 통합하여 Exp[A] 타입의 서브 클래스들로서 표현하고 있다. 서브 클래스들은 산술식에 대해서는 타입 변수 A가 Int로서 실례화 (instanciation) 되고, 논리식에 대해서는 Boolean 타입이 있도록 표현하고 있다. 이러한 표현법을 GADT (Generalized ADT) 라고 한다. 하스켈 또한 GADT 기능을 갖고 있다.

```
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
```

Fig. 1. ADT for AST in Haskell

```
enum Exp[A] {  // expressions
// GADT: A can be instanciated to Int and Boolean
 case V(x: String)    extends Exp[Int]
 case I(n: Int)       extends Exp[Int]
 case B(b: Boolean)   extends Exp[Boolean]
 case Add(x:Exp[Int], y:Exp[Int]) extends Exp[Int]
 case Sub(x:Exp[Int], y:Exp[Int]) extends Exp[Int]
 case Mul(x:Exp[Int], y:Exp[Int]) extends Exp[Int]
 case Eq(x:Exp[A], y:Exp[A])   extends Exp[Boolean]
 case Neq(x:Exp[A], y:Exp[A])  extends Exp[Boolean]
 case Lt(x:Exp[Int], y:Exp[Int]) extends Exp[Boolean]
}
enum Stm {     // statements
 case Ass(name: String, exp: Exp[Int])
 case Skip()
 case Comp(stm1: Stm, stm2: Stm)
 case If(bexp: Exp[Boolean], stm1: Stm, stm2: Stm)
 case While(bexp: Exp[Boolean], stm: Stm)
}
```

Fig. 2. ADT for AST in Scala

Fig. 2의 Stm 타입은 다섯 개의 문장들로 구성된 명령형 프로그래밍 언어 문장의 추상구문트리를 표현하고 있다. 문장들은 할당문(Ass), 문장의 연속 (세미콜론 ;) (Comp), 조건문(If), 반복문 (While) 이다. if 문은 논리식, then 에 소속된 문장, else 소속된 문장의 세 부분으로 구성된다. 보통 else 부분이 생략된 문법을 별도로 정의하기도 하는데, 여기서는 Skip 문을 사용함으로써 이를 대신할 수 있다. Skip()은 실행 결과 아무런 효과가 없는 문장의 의미를 갖는다. 문장 if bexp then statement else 은 else 부분이 생략된 것과 동일한 결과를 얻는다.

할당문 Ass은 문장의 왼쪽에 나타나는 변수 이름 V와 오른쪽의 Exp의 두 부분으로 구성된다. Comp는 stm1; stm2 형태의 두 문장의 연속 (합성)을 표현한다. While은 논리식 bexp와 이 값이 true 경우 수행할 문장 stm의 두 부분으로 구성되어 있다.

Fig. 3은 4 팩토리얼을 계산하는 명령형 언어 코드 및 추상구문트리를 코딩한 것이다 (결과 값은 변수 y에 남게 됨), 하스켈 코딩과 스칼라 코딩은 매우 유사하지만, 하스켈에서는 커리(Curried) 형태의 표현인 것에 비해 스칼라에서는 생성자(constructor)를 이용한 1차 함수식 형태의 표현을 사용하는 차이가 있다.

```
<Code with an imperative language>
x = 4;   y = 1; while (x != 1) {y = y*x; x = x-1; }

<Haskell code for AST>
fac4 =  Comp (Ass "x" (N 4))
         (Comp (Ass "y" (N 1))
           (While (Neg (E (V "x") (N 1)))
             (Comp (Ass "y" (Mul (V "y") (V "x")))
                   (Ass "x" (Sub (V "x") (N 1))))))

<Scala code for AST>
val fac4: Stm =
  Comp( Ass("x", I(4)),
    Comp( Ass("y", I(1)),
      While( Neq(V("x"), I(1)),
        Comp( Ass("y",Mul(V("x"), V("y"))),
              Ass("x", Sub(V("x"), I(1))) ))))
```

Fig. 3. Coding for 4!

2. Implementation of State Monad

2.1 Specification and Implementation of Monad

```
type ST = [(Var,Int)]
newtype State s a = State {runState :: s -> (a, s)}

class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
instance Monad (State s) where
  return a        = State (\s -> (a, s))
  State m >>= f   =
     State $ \s ->  let (a, s') = m s
                        State m' = f a
                    in  m' s'
get :: State s s
get = State $ \s -> (s,s)
set :: s -> State s ()
set s = State $ \_ -> ((),s)
```

Fig. 4. State Monad in Haskell

Fig. 4는 하스켈로 상태 모나드를 정의하는 모습이다. class에서 함수의 사양을 정의한 후 instance에서 (State s)타입의 특성에 따라 구현이 이루어진다. 이 모습은 자바의 Interface 나 스칼라의 trait 와 유사하다. 모나드의 구현을 위해서는 다음의 세 가지 요소가 표현되어야 한다.

(1) 고차함수 (람다 식 등으로 표현됨)

(2) 파라메터라이즈드(parameterized) 타입 (generics로 표현되는 Interface 혹은 trait),

(3) bind(>>=) 함수의 구현

List[A], Option[A] 등의 타입 정의에서처럼, 하나의 제너릭스를 갖는 경우, 이에 대한 모나드는 각각 Monad List, Monad Option 로서 구현된다.

2.2 Type Definition for State Transition

하스켈 타입 State s a 에서 s는 상태, a는 계산 결과 얻어지는 값의 타입을 의미한다. s -> (a, s) 은 함수 타입으로서, 상태 s1을 이용하여 계산 한 결과 a 타입의 값을 얻고, 상태는 s2로 바뀜을 의미한다. 명령형 프로그래밍 수행 과정에서 발생하는 변수들의 동적 변화는 상태 변환으로 설명된다. 이를 위한 상태는 하스켈 코딩 Fig. 4에서 type ST = [(Var,Int)] 으로, 스칼라 코딩 Fig. 5에서는 type ST = Map[String, Int] 의 타입이 정의되고 있다.

모나드 계산을 위해서는 a 타입의 값이 계산 결과로 드러나도록, State s 가 한 묶음으로 표현되어야 한다. 하스켈에서는 커리 (curried) 표현 식을 사용함에 따라 State s a ≡ (State s) a 이 성립하며, Fig. 4의 상태 모나드 정의는 instance Monad (State s) 형태로 이루어진다.

스칼라 타입에서는 커리 식을 사용하지 않으므로, Fig. 5에서 type StateS[A] = State[ST, A]를 이용하여 StateS 라는 이름의 타입을 정의하고, StateS를 기준으로 상태 모나드를 정의한다.

하스켈 (Fig. 4)의 모나드 컴비네이터는 바인드 (>>=)와 return, 그에 해당되는 스칼라 (Fig. 5)의 컴비네이터는 flatMap과 unit 이다. 사용되는 이름만 다를 뿐 동일한 기능을 한다. return과 unit은 어떤 한 값을 주어진 타입 문맥으로 래핑 (wrapping) 하는 기능을 한다.

스칼라로 표현되는 State 모나드에서 Int 타입의 값 10의 래핑은 생성자 State(s => (10, s)) 호출로서 구현된다.

2.3 Implementation of monad

스칼라 모나드 컴비네이터의 핵심은 flatMap이지만, for-yield 표현 (comprehension)을 이용하기 위해서 map 또한 함께 구현되어야 한다. map과 flatMap은 구현은 다음 형태로 구현할 수 있다.

(1) flatMap으로 map 기능을 대신 표현.

(2) join과 map을 구현하고, 이 둘로 flatMap을 표현.

Fig. 4에서는 바인드를 직접 정의하였고, Fig. 5에서는 두 번째 방법을 적용하였다. flatMap은 map 기능을 내포하고 있다. (list; List[Int]) 에 대해 다음이 성립한다.

list.map(x => x+2) == list.flatMap(x => List(x+2))

map의 매퍼 함수의 타입이 A => B 인 경우, 이 매퍼를 A => List[A] 형태로 변환하여 flatMap과 함께 사용하면, 이것은 원래의 map과 동일하다. 이 원리는 List뿐만 아니라 flatMap이 정의된 모든 타입에서 성립한다.

flatMap 이라는 용어는 두 함수 flat과 map의 합성 (flat . map)을 의미한다. 하스켈에서는 flatMap이 존재하지 않지만, 개념적으로 볼 때 그 타입은 flatMap :: (a -> m b) -> (m a -> m b) 라고 볼 수 있다. map은 map :: (a -> b) -> (m a -> m b) 의 타입을 갖는다. map의 첫 파라메터의 타입을 (a -> m b) 로 조정하면, map :: (a -> m b) -> (a -> m (m b)) 가 된다. 이 경우 flatMap의 결과 값의 타입이 (m b) 인 것이 비해, map은 (m (m b)) 인 차이가 있다. 따라서 map 함수의 결과 타입 (m (m b)) 의 값을 (m b) 타입의 값으로 변환하는 flattening 함수를 적용하면 flatMap과 map은 동일하게 된다. 즉, map의 실행과 그 결과 flattening하는 두 과정의 합성이 flatMap이다. 하스켈에서 이 함수를 join 이라고 부르고 있으므로, 본 연구의 스칼라 코딩에서도 동일한 이름을 사용한다.

이 상황은 제너릭스를 갖는 타입 m에 대해서 동일하게 적용되는 개념으로서, 모든 모나드의 구현에 이 원리를 적용할 수 있다. 예를 들어, 타입 m의 인스턴스 (instance)가 Option일 때 Option 모나드, StateS일 때 State 모나드가 된다.

2.4 join for State Monad

Fig. 5에서는 StateS에 대한 join 함수를 정의하고 있다. join 함수의 입력 파라메터는 StateS[StateS[A]] 타입의 값을 갖는 statesstatesa이며, 스칼라에서 객체지향 형태로 표현을 사용하기 위해 extension을 적용하였다. f(a, b) 형태의 함수적 표현을 a.f(b) 형태로 표현하기 위해서는 a가 속한 타입에 대하여 extension을 정의한다.

타입 StateS[A] 는 상태 전이를 이용하면서 계산하여 결과 값으로 A 타입의 값을 계산하는 것을 의미한다. StateS[StateS[A]] 은 연속된 두 번의 상태 전이를 의미하는데, join은 이를 StateS[A] 타입의 식으로 변환한다. Fig. 6의 ststi 는 StateS[StateS[Int]] 타입의 예이다. 이 식은 밖의 StateS 의 값 부분에 StateS[Int] 타입의 수식 State(sb => (100, s3)) 을 내포하고 있다. ststi를 join한 결과인 state1은 State(sa => (100, Map(c -> 3))) 가 된다. Fig. 5의 join 함수

의 정의는 이 과정을 표현하고 있다. statestatea에 outertr를 적용하여 안쪽에 내포된 StateS[A] 타입의 값 statea를 추출한 다음, runState(s1) 를 적용하여 튜플 (Int, ST) 타입의 값을 추출한다. 이 튜플을 다시 StateS(s => (,)) 로서 래핑함으로써 StateS[Int] 타입의 값을 만든다.

2.5 flatMap and map for State Monad

```
type ST = Map[String, Int]
case class State[S, +A](runState: S => (A,S))
type StateS[A] = State[ST, A]

def get: StateS[ST] = State((s: ST) => (s, s))
def set(s: ST): StateS[Unit] = State(_ => ((), s))

trait Monad[F[_]] {
  def unit[A](a: A): F[A]
  extension [A](fa: F[A])
    def flatMap[B](f: A => F[B]): F[B]
    def map[B](f: A => B): F[B]
}

object Monad {
  extension [A] (statestatea: StateS[StateS[A]])
    def join: StateS[A]  = statestatea match
      case State(outertr) =>
        State( (s: ST) =>
         { val (statea, s1) = outertr(s);
          statea.runState(s1)} )

  given stateMonad: Monad[StateS] with
   def unit[A](a:A): StateS[A] = State(s => (a, s))
    extension [A] (statea: StateS[A])
      def map[B](f: A => B): StateS[B] = statea match
        case State(tr) =>
          State((s: ST) =>
            { val (a, s1) = tr(s);
             (f(a), s1)} )
      def flatMap[B](f: A => StateS[B]): StateS[B] =
         statea.map(f).join
}
```

Fig. 5. State Monad in Scala

Fig. 5의 map에서는 StateS[A] 타입의 객체 states 의 래핑된 내용을 추출한 다음, 값 a에 주어진 A => B 타입의 함수 f를 적용하여 f(a)을 계산하고, 이것을 다시 State로서 래핑함으로써 StateS[A]를 얻는다. 이것은 카테고리 이론의 펑터(functor)를 구현하는 상황이다. 모나드를 구현하기 위해서는 먼저 펑터를 구현하는 것이 필요하다 [9]. State 펑터에서는 StateS[A] 타입인 객체에 대해서 map을 정의한다.

map과 join이 구현되면 flatMap은 공짜로 얻을 수 있다. map의 파라메터인 A => B 타입의 매퍼 함수 f 대신, A => StateS[B] 타입의 매퍼 함수를 적용하여 map을 계산하면 그 결과 StateS[StateS[A]] 타입의 값을 얻는데, 여기에 join을 적용함으로써 StateS[A]를 얻게 된다.

flatMap은 join의 기능도 내포하고 있다. flatMap의 파라메터인 매퍼 함수가 identity (x => x) 인 경우, flatMap은 join과 동일하다. 예를 들어, Option[Option[Int]]를 Option[Int] 로 flattening 하는 기능을 join 대신 다음과 같이 flatMap 으로 표현할 수 있다.

Some(Some(10)).flatMap(x => x) == Some(10)

```
var s2: ST =  Map("b" -> 2)
var s3: ST =  Map("c" -> 3)
val sti: StateS[Int] = State(sa => (10, sa))
val ststi: StateS[StateS[Int]] =
   State(sa => (State(sb => (100, s3)), s2))
val state1: StateS[Int]= ststi.join
    // State(sa => (100, Map(c -> 3)))
val st1: ST => (Int, ST) = state1.runState
    // sa => (100, Map(c -> 3))
val tpl: (Int, ST) = st1(s1)   // (100, Map)
```

Fig. 6. Example of State monad and join function

3. An Evaluator By Applying State Monad

3.1 for-yield comprehension for flatMap

```
case class Id[A](runId: A) {
  def map[B](f: A => B): Id[B] = Id(f(runId))
  def flatMap[B](f: A => Id[B]): Id[B] = map(f).join
}
extension [A](idida: Id[Id[A]])
  def join: Id[A] = idida.runId

def main(args: Array[String]): Unit =
  val res1 =
    Id(10).flatMap(a =>
      Id(30).flatMap(b =>
        Id(100).map(c => (a+b+c) )))   // Id(140)
  val res2 = for {
    a <- Id(10)
    b <- Id(30)
    c <- Id(100)
  } yield (a+b+c)   // Id(140)
```

Fig. 7. Identity Monad and for comprehension

Fig. 7은 Identity 모나드와 for-yield 표현을 소개하고 있다. Identity 모나드는 매우 단순하며, 아무런 효과 (effect)를 표현하지 않는다. 여기서는 모나드의 실험 및 for-yield 표현을 목적으로 사용되고 있다. join과 map을 정의한 후, 이를 이용하여 flatMap을 정의한다.

res1은 두 번의 flatMap과 마지막 map을 적용하여 계산하고 있다. 이처럼 flatMap 의 체인과 map에 의한 마무리는, for-yield 로서 대신 표현할 수 있으며, res1은 res2와 동일한 값을 갖는다. for-yield는 flatMap의 구문적 간결성 (syntactic sugaring) 목적의 표현일 뿐, 그 밖의 다른 의

미를 갖지는 않는다.

스칼라의 flatMap에 대한 for-yield 표현은 하스켈의 bind에 대한 do 구문의 표현과 비교될 수 있다.

3.2 Evaluator with pure-functional style

Fig. 8에 정의된 eval 함수는 Fig. 2에 소개된 Exp[A] 타입의 수식을 계산하는 계산기이다. 이 함수의 출력 타입 A는 산술식인 경우 Int이고, 논리식인 경우 Boolean이다.

수식은 변수를 포함하고 있으므로, 변수를 계산하기 위해서는 ST 타입의 객체 s가 주어져야 한다. ST는 Fig. 5에 정의된 것처럼 key-value의 타입 type ST = Map[String, Int]으로 표현된다.

```
extension[A] (exprt: Expr[A])
  def eval(s: ST): A = exprt match
    case V(x)      => s(x)
    case I(n)      => n
    case B(b)      => b
    case Add(x,y)  => x.eval(s) +  y.eval(s)
    case Sub(x,y)  => x.eval(s) -  y.eval(s)
    case Mul(x,y)  => x.eval(s) *  y.eval(s)
    case Eq(x,y)   => x.eval(s) == y.eval(s)
    case Neq(x,y)  => x.eval(s) != y.eval(s)
    case Lt(x,y)   => x.eval(s) <  y.eval(s)
```

Fig. 8. Evaluator for expression in a pure function

3.3 Evaluation of expressions with state monad

Fig. 8의 eval 함수에서는 상태 s를 명시적으로 파라메터로서 패스하는 방법을 적용하고 있지만, 모나드를 적용하여 코딩한 Fig. 9에서는 이 과정이 표현되지 않는다. Fig. 8의 코딩은 모든 필요한 정보 (예를 들어 상태 s)를 드러내어 표현하는 평면적 구조인 반면에, 모나드 프로그래밍은 코드를 상부 구조와 하부 구조로 추상화한 형태로 구성한다. 타입의 특성에 따라 루틴하게 처리하는 과정은 하부 구조의 모나드에서 처리하고, 상부 구조에서는 값에 대한 계산만을 표현한다.

Fig. 9 의 코딩에서 상태 변환 과정은 모나드 컴비네이터인 bind에 의해서 처리된다. do는 바인드 함수의 적용을 의미하는데, (State s) 에 대한 bind 함수 정의에 따라 상태 변환이 자동으로 처리되고 있으며, 상부 구조에서는 상태 변환에 관심을 가질 필요가 없다. 예를 들어, (Add a1 a2)를 처리할 때, 의미적으로 상태에 대한 정보가 필요하지만 (변수가 포함될 수 있으므로), 상부 구조인 do 에서는 하부 구조인 모나드에 의해서 처리되어 얻어진 두 Int 값 x와 y의 더하기 (x+y) 만을 표현하고 있다.

```
aEvalM :: Aexp -> ST -> Int
aEvalM aexp initST =
  fst $ runState (execAexp aexp) initST
execAexp :: Aexp -> State ST Int
execAexp (N n)   = return n
execAexp (V var) =
   get >>= \s -> return (fromJust $ lookup var s)
execAexp (Add a1 a2)  = do  { x <- execAexp a1
                            ; y <- execAexp a2
                            ; return (x+y)
                            }
execAexp (Mul a1 a2)  = do  { x <- execAexp a1
                            ; y <- execAexp a2
                            ; return (x*y)
                            }
execAexp (Sub a1 a2)  = do  { x <- execAexp a1
                            ; y <- execAexp a2
                            ; return (x-y)
                            }
```

Fig. 9. Evaluator using state monad in Haskell

Fig. 9에서 계산기 함수 aEvalM의 실제 계산은 execAexp에서 이루어진다. 이 결과 얻어지는 값은 (State ST) Int 타입의 객체인데, 우리가 원하는 값은 Int이므로, aEvalM에서 이 객체 내에 있는 튜플 (a, st) 중 a 값을 추출하는 fst 함수를 적용한다. aEval과 execAexp는 오직 산술식 만을 다루므로, 논리식에 대한 함수 bEval과 execBexp가 별도로 정의되어야 한다. 이 함수 또한 유사한 방법으로 코딩될 수 있으며, 여기서는 생략하기로 한다.

```
extension[A] (exprst: Exp[A])
 def eval(s: ST): A = execExp.runState(s)._1
 def execExp: StateS[A] = exprst match
  case V(x)     => for {s <- get } yield(s(x))
  case I(n)     => unit(n)
  case B(b)     => unit(b)
  case Add(x,y) =>
    for {a <- x.execExp; b <- y.execExp} yield (a+b)
  case Sub(x,y) =>
    for {a <- x.execExp; b <- y.execExp} yield(a-b)
  case Mul(x,y) =>
    for {a <- x.execExp; b <- y.execExp} yield(a*b)
  case Eq(x,y)  =>
    for {a <- x.execExp; b <- y.execExp} yield(a==b)
  case Neq(x,y) =>
    for {a <- x.execExp; b <- y.execExp} yield(a!=b)
  case Lt(x,y)  =>
    for {a <- x.execExp; b <- y.execExp} yield(a<b)
```

Fig. 10. Evaluator using state monad in Scala

Fig. 10의 코드는 Fig. 9의 하스켈 코드와 동일한 구조를 갖는 스칼라 코드이다. 하스켈의 aEvalM은 스칼라의 eval과 같은 구조이며, execAexp는 execExp와 유사하다

(execExp는 논리식에 대한 계산도 함). 하스켈의 do는 스칼라의 for-yield 에, return은 unit에 해당된다.

3.4 Evaluation of statements with state monad

명령형 프로그래밍언어의 여러 의미론 중, 본 연구에서는 동작 의미론(Operational Semantics) [10] 을 따라 인터프리터를 구현하였다. 프로그램 실행 과정에서 발생하는 변수들 값의 동적 변화는 상태(state)라는 개념으로 설명된다. 각 문장이 실행됨에 따라 한 상태가 변화하고, 변화된 상태는 그 다음 문장 실행에 사용되는 과정이 반복되고 있다. 각 문장은 상태 변환(state transition)으로 해석되며, 두 문장의 연속적 수행은 두 상태 변환의 합성(composition)이 된다. 두 상태 변환 s1 => s2 과 s2 => s3의 합성 결과는 상태 변환 s1 => s3 이 된다. 이 방법에 따라, 명령형 프로그램은 어떤 주어진 초기 상태로부터 프로그램 수행 종료 시의 상태로 변환으로 해석된다.

```
evalStm :: Stm -> ST -> ST
evalStm stm initST  -- The result is stored in state
  = snd $ runState (execStm stm) initST

execStm :: Stm -> State ST ()
execStm (Ass var aexp)  =
    do { s <- get
       ; v <- execAexp aexp
       ; let s' = update var v s
       ; set s'
       }
execStm (Comp stm1 stm2)   =
    do { execStm stm1
       ; execStm stm2
       }
execStm Skip  = State (\s -> ((), s))
execStm (If be stm1 stm2)  =
    do { b <- execBexp be
       ; if b then execStm stm1 else execStm stm2
       }
execStm (While be stm) =
    do { b <- execBexp be
       ; if b  then execStm (Comp stm (While be stm))
              else execStm Skip
       }
```

Fig. 11. Evaluator of Statements using monad in Haskell

Fig. 11과 Fig. 12의 evalStm은 문장 Stm과 초기 상태 ST를 입력 받아, Stm 실행 결과 얻어지는 최종 상태 ST를 출력한다 (동일한 원리를 두 언어로 구현함). evalStm의 실질적인 수행은 execStm에서 상태 모나드를 적용하여 이루어진다. 이 상황은 Exp 계산기인 Fig. 9, Fig 10과 유사하다. Exp의 계산 결과는 값(value)으로서, 이는 상태 객체 (s -> (a, s)) 중 a에 해당되므로, Fig. 9에서 fst $ runState (execAexp aexp) initST 으로서 이 값을 추출하지만, Fig. 11과 Fig. 12에서는 상태 s가 목표이므로, snd $ runState (execStm stm) initST 이라는 표현을 사용하는 차이점이 있다.

문장 Stm은 Ass(할당문), Comp (세미콜론 ; 에 대한 두 문장의 연속), Skip, If(조건문), While(반복문)으로 구성된다. Ass 실행은 모나드 하부 구조에 있는 현재 상태 값을 끌어올려 (s <- get), 산술식의 값을 계산하고 상태를 수정한 다음, 다시 모나드 하부 구조의 상태에 넣는다 (set s'). Skip 문의 실행은 상태를 변환시키지 않는다.

두 문장 stm1과 stm2이 주어진 Comp에서는, stm1을 먼저 execStm으로 실행하고 (이때 모나드 하부 구조의 ST 상태가 변환됨), 그 변화된 ST 상태를 이용하여 다시 stm2를 execStm으로 실행하며, 이 결과 변환된 ST 상태가 두 문장 연속의 실행 결과가 된다.

If 문의 실행은 각 구현 언어의 If를 이용하여 처리한다. stm1과 stm2 둘 중 하나가 execStm으로 수행된다.

While 문은 조건식 be의 값이 true 동안 반복해서 stm을 execStm으로 수행하면서 상태를 생성해 낸다 (모나드의 하부구조에서 이루어짐). 반복적 수행을 위해 stm 문과 while 문을 Comp로 묶는 새로운 구문을 만드는 방법을 적용한다.

```
extension (stm: Stm)
 def evalStm(s: ST): ST = execStm.runState(s)._2
 def execStm: StateS[Unit] = stm match
   case Ass(name, exp) =>
     for { s <- get; v <- exp.execExp;
           r <- set(s.updated(name, v))
         } yield (r)
   case Skip() => State(s => ((), s))
   case Comp(stm1, stm2) =>
     for { s1 <- stm1.execStm;
           r <- stm2.execStm
         } yield(r)
   case If(bexp, stm1, stm2)  =>
     for { b <- bexp.execExp;
           s <- if b then stm1.execStm
                     else stm2.execStm
         } yield (s)
   case While(be, stm) =>
     for { b <- be.execExp;
           s <- if b then
                  Comp(stm, While(be, stm)).execStm
                else Skip().execStm
         } yield (s)
```

Fig. 12. Evaluator of Statements using monad in Scala

Fig. 3의 fac4를 초기 상태 Map()과 함께 evalStm으로 실행하여 얻어지는 상태를 출력하면 다음과 같다.

println(fac4.evalStm(Map()))

화면 출력 결과: Map(x -> 1, y -> 24)

IV. Comparison of Haskell with Scala

하스켈은 다음과 같은 점에서 스칼라와 차이가 있다. 첫째, 커리적으로 (Currried) 고차함수를 표현하기 쉽다. 예를 들어, 1 더하기 2의 커리적 표현은 하스켈에서 괄호 없이 (add 1 2) 로, 스칼라에서는 괄호가 필수적인 (add (1) (2)), 스칼라가 편리성과 가독성 측면에서 불리하다. 대략적으로, 하스켈에서는 고차 함수적 표현을, 스칼라에서는 일차 함수적 표현을 적용하는 경향이 높다. 타입 정의 또한 하스켈에서는 커리적 표현을 사용하지만, 스칼라의 타입에서는 커리적 표현을 사용하지 않으므로, 타입 변수가 두 개 이상인 경우, Fig. 5 에서처럼 type 을 사용한 래핑 과정을 거쳐야 하는 번거로움이 있다.

둘째, 하스켈에서는 함수의 순수성/비순수성이 타입으로 표현되고, 컴파일러의 타입 체킹에 의해서 이것이 검증된다. 일반 함수는 모두 순수 함수이고, IO 타입을 갖는 함수는 비순수 함수이다. 함수의 순수성을 유지하기 위해, 타입 체킹으로 순수 함수가 비순수 함수를 호출할 수 없도록 한다. 그러나 스칼라에서는 함수의 부작용(side-effect) 기능을 배척하지 않으며, 함수의 순수성/비순수성을 체크하거나 방지할 수 있는 기능이 없다. 스칼라에서 함수의 순수성을 표현하는 것은 오직 사용자의 몫이다.

셋째, 하스켈의 모나드 프로그래밍은 모나드 이론을 충실히 따르고 있다. 모나드의 전제 조건으로서, 함수 합성의 결합법칙이 요구되며, 하스켈은 이를 만족한다.

$$(f \cdot g) \cdot h = f \cdot (g \cdot h)$$

스칼라에서는 모든 함수를 순수 함수로서 정의하지는 않으므로, 이 결합법칙은 상황에 따라 성립되지 않는다.

그럼에도 불구하고, 실용적으로 이것이 큰 문제가 되지는 않는다. 필요에 따라 사용자가 주의하여 코딩함으로써 함수의 순수성과 위의 모나드 조건을 만족할 수 있다. 또한, 위의 모나드 조건을 만족하지 못하는 경우라도 모나드 함수 flatMap을 적용할 수 있다.

실용적으로는 flatMap 대신 for-yield 형태의 표현이 중요하며, for-yield에서는 map 함수의 정의가 필수적이다. 본 연구에서는 join 함수와 map을 구현함으로써, flatMap을 자연적으로 구하는 방법을 적용하였다.

V. Conclusions

본 연구에서는 스칼라의 함수형 프로그래밍 기능을 실험하기 위해, 함수형 프로그래밍 기법을 적용하여 명령형 프로그래밍 언어의 인터프리터를 개발하였다. 스칼라는 람다 식, 제너릭스, 모나드 등의 대수적 데이터 타입 등 주요 함수형 프로그래밍의 기능을 충분히 잘 표현하고 있다고 보여 진다. 이러한 스칼라의 특성은 코틀린과 자바에도 영향을 줄 것이다. 함수형 프로그래밍 기법은 점차 그 영역을 확대해 나가고 있으며, 보편화되고 있다. 웹의 React, callback, Promise 등에 고급 함수형 프로그래밍이 적용되는 것이 그 좋은 예이다.

본 연구에서 적용한 하스켈과 스칼라 프로그래밍의 연관성은 함수형 프로그래밍을 공부할 때 좋은 방법이 될 수 있다. 함수형 프로그래밍은 이론을 다루게 되는데, 하스켈은 이론을 비교적 쉽고 자연스럽게 표현할 수 있으므로, 하스켈을 이용하여 함수형 프로그래밍을 이해하고, 이 개념을 다른 언어에 적용하는 것이 좋은 방법이 될 수 있다.

REFERENCES

[1] M. Odersky, and et al, Programming in Scala, Fifth Edition, artima, June 2021.

[2] M. Pilquist, and et al, Functional Programming in Scala, 2nd Edition, Manning, November 2022.

[3] R. Urma, and et al, Modern Java in Action: Lambdas, streams, functional and reactive programming. 2nd Edition, Manning, November 2018.

[4] M. Vermeulen, and et al, Functional Programming in Kotlin, Manning, October 2021.

[5] Haskell homepage, https://www.haskell.org/

[6] Sugwoo Byun, “Interpreters for an Imperative Language Using State Monad”, Journal of KIIT, Vol. 12, No. 2, pp. 145-152, Feb. 2014.

[7] S. Peyton Jones and P. Wadler, “Imperative Functional Programming”, 20'th Symposium on Principles of Programming Languages, ACM Press, 1993.

[8] Sugwoo Byun, “Implementation of Nondeterministic Compiler Using Monad”, Journal of The Korea Society of Computer and Information, Vol. 19, No. 2, pp. 151-159, Feb. 2014.

[9] Sugwoo Byun and Woo Gyun, “Functor and Monadic Programming in Haskell”, Communications of the Korean Institute of Information Scientists and Engineers, Vol. 35, No. 3, pp. 33-40, March. 2017

[10] Operational Semantics, https://en.wikipedia.org/wiki/Operational_semantics

Author

Sugwoo Byun received the B.S. and M.S.. degrees in Computer Science from Soongsil University, Korea, in 1980 and 1982, respectively. He received Ph.D. degree in Computer Science from the University of East Anglia, UK, in 1994. Dr. Byun had been a Principal Researcher at ETRI from 1982 to 1998, and has been working as a Professor of Computer Science at Kyungsung University, Busan, Korea, since 1999. He is interested in theories of programming languages, functional and concurrent programming, and formal proofs.