모나드를 이용한 어셈블리 언어 인터프리터 개발
(Development of an Assembly Language Interpreter Using Monad)

변 석 우 †
(Sugwoo Byun)

요 약 하스켈의 모나드는 순수 함수형 프로그래밍뿐만 아니라 명령형 형태의 프로그래밍도 가능케 하고 있다. 본 연구에서는 순수 함수형 프로그래밍 방식으로 코딩된 어셈블리 언어 인터프리터 구현을 모나드 방식으로 재 구성함으로써 모나드 추상화와 프로그래밍 기법의 특성을 부각시킨다. 모나드 프로그래밍은 스택과 심볼 테이블에 상태 모나드를 적용하는 것과, 이 모나드들을 상태 모나드 트랜스포머를 이용하여 통합 구성하는 두 단계로 이루어진다. 결과적으로, 모나드 프로그래밍에 의한 코드는 순수 함수형 스타일의 코드보다 훨씬 더 간결하고 직관적임을 볼 수 있다.
키워드 : 하스켈, 모나드, 모나드 트랜스포머, 상태 모나드, 상태 모나드 트랜스포머

Abstract Monad in Haskell allows one to do imperative-style programming as well as pure functional programming. In this work, we characterize monadic abstraction and its programming technique by restructuring an assembly language interpreter coded in pure functional style into the one by the monadic style. Monad programming consists of two phases; the State monad is applied to a stack and a symbol table, and then a State Monad Transformer integrating these two monads is constructed. As a result, we can see that the program code by monad programming is much clearer and more intuitive than one written in the pure functional style.
Key words : Haskell, Monad, Monad Transformer, State Monad, State Monad Transformer

1. 서 론

등식 추론(Equational Reasoning)의 원리는 하스켈과 같은 순수 함수형 언어가 갖는 가장 핵심적인 특징이다. 그러나, 프로그래밍의 실제에 있어서는 입출력처럼 등식 추론이 아닌 부대적 효과(side-effect)를 이용하는 프로그래밍을 적용해야 할 경우가 있다. 등식 추론의 원리와 부대적 효과 프로그래밍의 원리를 함께 유지하는 것은 매우 어려운 문제였으나, 1989년 Moggi가 모나드 이론을 제시하면서 문제 해결의 돌파구가 열리게 되었다[1]. Wadler가 이 이론을 구현 모델로 제안하고[2], 그것이 성공적으로 구현되어 하스켈 1998년 표준 버전에 채택되었다. 현재 모나드는 하스켈의 가장 대표적 특징으로 주목받고 있다.

여러 모나드 중에서 가장 대표적인 것은 상태 모나드(State Monad)이다. 본 연구에서는 스택 머신 용 어셈블리 언어 인터프리터를 상태 모나드로서 구현하는 과정을 설명한다. 순수 함수형(side-effect free) 프로그래밍 방식과 모나드 방식을 비교 설명함으로써 모나드 프로그래밍의 특징을 명확히 설명하고자 한다.

모나드 방식에 의한 프로그래밍에는 세가지 특성이 고려된다. 첫째, 각 어셈블리 명령어의 수행 결과가 스택에 저장되므로 스택의 상태가 변화된다. 이 과정은 (State Stack a) 모나드로서 표현된다. 둘째, 이 어셈블리언어의 변수는 동적으로 변화되므로 이를 관리하기 위한 심볼테이블이 필요한데, 이를 위해 (State Symtab a) 모나드가 이용된다. 마지막으로 여러 모나드들을 합성하는 모나드 트랜스포머(Monad Transformer) 기법이 필요한데, 여기서는 상태 모나드 트랜스포머(State Monad Transformer)인 StateT를 이용하여 이 두 상

· 이 논문은 2010학년도 경성대학교 학술연구비지원에 의하여 연구되었음
† 정 회 원 : 경성대학교 컴퓨터학부 교수
swbyun@ks.ac.kr
논문접수 : 2010년 1월 25일
심사완료 : 2010년 2월 20일

정보과학회논문지: 소프트웨어 및 응용 제37권 제5호(2010.5)

태 모나드를 연동시키는 프로그래밍 기법을 이용한다. 한편, 어셈블리 언어의 Goto 기능의 구현 기법에 대해서도 논의한다.

본 연구에서는 제2절에서 상태 모나드의 특성을 간단히 소개하고, 제3절에서 스택 기반의 어셈블리 언어 인터프리터의 구현을 순수 함수형 방법과 모나드 방법으로 구성하고 비교함으로써 모나드 추상화(abstraction)와 모나드 프로그래밍 기법의 특징을 설명한다. 소수의 명령어들의 구현으로부터 시작하여, 점진적으로 기능이 추가된 인터프리터 구현을 비교 분석함으로써 프로그래밍 기법을 명확히 설명하도록 하였다. 제4절에서는 본 연구에서 모나드/모나드 트랜스포머에 의한 프로그램 코드가 갖는 모듈성(modularity)에 대해서 설명하고 있다. 모나드 추상화에 의한 코드는 순수 함수형 스타일의 코드보다 훨씬 더 간결하고 명확한 모습을 보여준다. 제5절에서는 관련 연구와 함께 이 연구의 의미를 정리한다.

이 논문은 독자들이 하스켈에 대한 기본 지식과 이해를 하고 있음을 전제로 작성되었다. 하스켈에 경험이 없는 독자는 [3,4]와 하스켈 홈페이지[5]의 정보를 참조하기 바란다.

2. 하스켈의 모나드

2.1 모나드 기본

하스켈에서는 계산과 값을 타입으로서 구분하여 표현한다. 모나드로 정의된 컴퓨터 프로그래밍의 한 계산(computation) m이 계산을 수행한 결과 a 타입의 결과를 얻는 상황은 (m a)로 표현된다. 계산 m은 응용 분야의 특성에 따라 입출력 프로그래밍, 동시성(concurrency) 프로그래밍, 예외처리(Exception Handling), 컴파일러의 파서, 등식 추론이 적용되는 리덕션 등등 다양하게 존재한다. 예를 들어, IO Int, IO a, Maybe Int, State Stack Int 등이 있다. (IO Int)는 입출력을 수행하여 그 결과로서 정수 값을 얻고 있음을 의미한다. 만약 입출력의 결과 얻어지는 임의의 값인 경우는 타입 변수를 사용하여 (IO a)로 표현된다. 계산의 다양성에도 불구하고 하스켈에서는 모든 계산을 모나드 인터페이스에 의하여 통일적으로 정의하는 특징을 갖는다.

모나드를 정의하기 위해서는 먼저 m에 해당되는 타입이 파라메터로서 타입을 갖는 타입 구성자(Constructor)가 되도록 정의되어야 한다. 예를 들어, data Maybe a = Just a | Nothing 로 정의되는 Maybe는 타입 a를 파라메터로 갖는 타입 구성자이다.

일단 타입이 정의되면, 그 타입의 특징이 반영되어 계산되는 모나드를 정의할 수 있다. 하스켈의 모나드 인터페이스는 Monad 클래스 이용하여 표현된다. 이 클래스에는 다음 그림 1과 같이 두 개의 오퍼레이터 return과 >>=를 포함하고 있다.

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

그림 1 모나드 클래스와 오퍼레이터

각 계산 m에 대한 모나드는 위의 두 오퍼레이터에 대한 instance를 정의함으로써(즉, overloaded 함수 정의) 이루어진다. 예를 들어, Maybe 타입은 계산 과정에서 실패가 발생하였는지의 여부를 판단하기 위해서 이용된다. 계산 과정에서 단 한번이라도 실패(Nothing으로서 표현됨)가 발생하였다면 전체 결과는 실패가 되며, 모든 과정에서 실패가 없을 경우에만(Just로서 표현됨) 정상적인 계산으로 인정하도록 하는 원리를 적용한다. 이 원리에 따라 정의되는 Maybe 모나드는 다음 그림 2와 같다.

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

그림 2 Maybe 모나드의 정의

return 오퍼레이터는 아무런 역할을 하지 않고 단지 주어진 값을 모나드 형태로 투입(inject)하는 역할을 하며, >>= 오퍼레이터는 두 계산이 주어졌을 때, 첫 단계의 계산 결과를 두 번째 단계로 입력 변수에 바인드하여(따라서 '바인드(*bind*)'라고 불림) 이 둘을 순차적으로(sequential) 수행되도록 하는 기능을 한다. 모든 계산 과정은 이 바인드의 반복적 적용이므로, "계산 과정의 단 한번의 실패가 전체의 실패"로 인정되는 Maybe 계산의 기본 원리가 이 바인드의 정의를 통해서 반영되고 있음을 볼 수 있다.

Maybe 모나드를 이용하는 '안전한 더하기' 함수 add는 다음과 같이 정의될 수 있다.

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

바인드 오퍼레이터 대신 치장된 구문(sugared syntax)인 do를 이용하여 다음과 같이 표현된다.

```
add x y = do a <- x
             b <- y
             return (a + b)
```

2.2 State 모나드

State 모나드를 위해 정의되는 타입은 newtype State s a = State (s -> (a,s)) 로서 정의된다. State는 상태 값을 표현하는 변수 s와 계산된 값을 표현하는 변수 a의 두 파라메터를 갖는다. 여기서 s는 여

러 타입으로 실례화될 수 있다. 타입(State Stack a)는 하부구조에서 스택 상태를 변화시켜 가면서 계산하고 결과로서 a 타입의 값이 출력됨을 의미한다. (State Symtab a)는 하부구조에서 심볼 테이블의 상태가 변화하는 모습으로 해석된다.

상태의 변환은 타입 s -> (a, s)로 표현되는데, 이것은 State 타입의 계산에 의해서 어떤 한 상태 s_1이 s_2로 변이되고, 계산 결과 a가 출력되는 것을 뜻한다.

(State s a)는 ((State s) a)의 간략한 표현이므로, State에 대한 모나드 정의는 다음 그림 3과 같이 (State s)에 대해서 return과 >>= 를 정의하는 것에 해당된다.

```
instance Monad (State s) where
  return v       = State (\s -> (v,s))
  State m >>= f  = State $ \s -> let (x,s1) = m s
                                 State n = f x
                                 in (n s1)
```

그림 3 State 모나드 정의

return과 >>=의 기능은 그림 4와 같이 설명될 수 있다. (return :: a -> State s a)에서는 하부 구조에서 상태의 변환이 일어나지 않으며 입력되는 값을 단지 상부구조에 전달하는 역할을 한다.

그림 4 상태 모나드에서의 return

바인드는 ((>>=) :: State s a -> (a -> State s b) -> State s b) 타입을 갖는다. 바인드에 두 개의 계산이 주어졌을 때, 첫 계산의 계산 결과의 타입과 나머지 다른 계산의 입력의 타입이 일치하면 이 두 계산은 바인드에 의해 합성될 수 있다. 그림 5의 왼쪽 그림은 (State s a)의 타입을 가지며, 계산 결과 (x::a, s_1::s)가 출력됨을 보여 준다. 두 번째 계산은 (a -> State s a)의 타입으로서 표현되는데, a 타입의 값과 하부 구조의 상태를 입력 받아 처리하여 값 (y::a, s_2::s)를 출력한다. 왼쪽의 출력과 오른쪽의 입력 타입이 일치하므로 이 두 계산은 바인드에 의해서 순차적으로(sequentially) 합성될 수 있다.

그림 3 (>>=)의 정의를 분석해 보면, \s -> let (x,s_1) = m s는 상태 s를 이용하여 첫 계산결과 (x, s_1)를 얻는다. 두 번째의 계산은 (f x)와 (n s_1)의 두 단계에 걸쳐 수행된다. 첫 계산에서 얻어진 x::a를 이용하여 상부 구조의 계산 (f a)를 수행하여 (State n) :: (State s b)을 얻는다. 패턴 매칭에 의해서 구성자

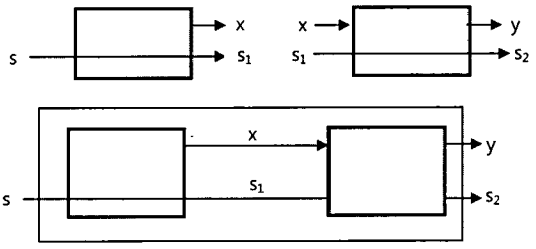

그림 5 바인드에 의한 계산의 연속

State를 벗겨 내면 n은 s -> (a, s) 타입의 형태를 갖게 되는데, (n s_1)의 수행 결과 ((y,s_1)::(b,s))를 얻는다. 이 과정을 전체적으로 종합하면, >>= 정의의 오른쪽 부분은 (State $ \s -> (y, s_1))::(State s b)가 된다. 타입을 보면 하부구조의 상태는 항상 같은 타입을 갖고 있음을 알 수 있다.

State 모나드 유형의 계산은 빈번하게 발생하는데, Parser와 IO 모나드 역시 여기에 해당된다. Parser 모나드에서 상태 s는 입력 스트링이며, IO 모나드에서 s는 시스템의 수행 시스템 환경인 World가 된다.

2.3 상태 모나드 트랜스포머 StateT

상태 변환을 위한 타입은 s -> (a, s)으로서, 상태 s를 변화시켜 가면서 값 a를 계산해 냄을 의미한다. 상태 모나드를 확장시켜, 상태 모나드가 어떤 한 내장된 모나드 m을 이용하면서 계산하는 현상을 표현하는 타입은 s -> m (a, s)로서, 이것이 상태 모나드 트랜스포머의 핵심이다. 상태 모나드 트랜스포머를 위한 타입은 newtype StateT s m a = StateT (s -> m (a,s)) 로 정의된다.

상태 모나드 트랜스포머로의 확장에서와 같이, 일반적으로 어떤 한 모나드를 모나드 트랜스포머로 확장할 때는 트랜스포머 타입 정의에 내장 모나드 m을 파라메터에 추가한다. 또한 관례적으로 트랜스포머의 이름에는 모나드와 차별될 수 있도록 모나드 이름 뒤에 T를 덧붙인다.

상태 모나드 트랜트포머 또한 모나드이다. 타입 (StateT s m)에 대해서 Monad 클래스의 인스턴스를 정의함으로써 StateT 모나드가 정의되는 데, 이것은 다음 그림 6과 같이 정의되어 있다.

```
instance (Monad m) => Monad (StateT s m) where
  return v         = StateT $ \s -> return (v,s)
  StateT t >>= f   = StateT $ \s -> do (a, s') <- t s
                                       (StateT f') <- return $ f a
                                       f' s'
```

그림 6 StateT 모나드 정의

그림 3의 State 모나드와 비교할 때, StateT의 return과 바인드 정의(오른쪽 부분)에는 추가적으로 return과 do가 사용되고 있음을 볼 수 있다. 간단히

설명하여, 이것은 내장된 모나드 m의 계산이 적용되고 있음을 의미한다.

모나드와 모나드 트랜스포머에 대한 자세한 내용은 하스켈 홈페이지[5]의 검색기능을 통해서 얻을 수 있으며, 또한 [6-8]을 참조하기 바란다.

3. 스택 어셈블리 언어 인터프리터의 구현

3.1 스택 어셈블리 언어와 추상 머신

그림 7처럼 하스켈로서 표현되는 스택 어셈블리 언어와 추상 스택 머신을 가정한다. (이들은 [9, Chapter 2]에서 소개되는 스택 어셈블리 언어의 부분집합으로서 그들과 동일한 동작원리를 갖는다).

```
type Prog = [Code]
data Code = Push Int | Plus | Mult | Gt
          | Rvalue Name | Lvalue Name | Assgn
          | Lab Int | Goto Int | GoFalse Int
type Stack = [Int]
type Symtab = [(String,Value)]
```

그림 7 스택 어셈블리 언어와 추상 머신

이 언어를 수행하는 추상 머신은 프로그램을 저장하는 Prog, 데이터를 저장하고 연산하는 Stack, 그리고 프로그램 수행 중 변수들의 상태를 표현하는 Symtab의 세 기억 장치를 가지고 있다. 하스켈에서 이들은 모두 리스트로 표현되며, Symtab은 튜플(String, Value)의 리스트이다. 이 명령어들은 특성에 따라 다음과 같이 네 개로 분류된다.

- 스택 내용 접근 - Push
- 연산작용 - Plus, Mult, Gt(Greater Than)
- 변수사용 - Rvalue, Lvalue, Assgn
- 제어작용 - Lab, Goto, GoFalse

스택과 연산작용을 위한 명령어들의 동작의미는 다음과 같이 집합 Prog × Stack → Prog × Stack의 리덕션(reduction) 관계로 표현된다.

<Push n : ops, s> → <ops, n:s>

<Plus : ops, n:m:s> → <ops, (n+m):s>

<Mult : ops, n:m:s> → <ops, (n*m):s>

<Gt : ops, n:m:s> → <ops, (m>n):s>

여기서 Gt를 구현하는 연산자 >는 편의상 > :: Int -> Int -> Int 의 타입을 갖도록 정의하며, False 대신 0, True 대신 1로 표현하기로 한다. 필요에 따라 산술 연산자들을 추가 할 수 있으며, 이들의 동작의미는 위와 유사한 방법으로 정의할 수 있다.. 모든 연산자들의 계산은 스택의 탑에서 발생하는데, 스택에 있는 데이터들을 접근할 때 묵시적으로 스택에서 Pop이 발생함을 알 수 있다.

예를 들어, 산술식 (1 + 2) * 3은 후연산 표현(postfix notation)에 의해 1 2 + 3 * 표현되며, 이들은 쉽게 다음과 같은 명령어들로서 표현된다 - [Push 1, Push 2, Plus, Push 3, Mult]. 이 명령어들을 계산(evaluation)하면 결과 값으로 9를 얻는다.

3.2 하스켈을 이용한 산술 연산자들의 구현

3.2.1 순수 함수형 방식에 의한 구현

그림 8처럼 스택 명령어를 계산하는 함수 eval :: Prog -> Int 의 수행은 exec 함수가 Prog에 있는 각 명령어들을 반복 수행함으로써 처리된다. exec :: Prog -> Stack -> Stack 은 스택을 이용하여 주어진 프로그램을 계산한다. 계산 시작 시 스택의 초기 값은 []이며, 계산이 완료되면 그 결과 값은 스택의 탑에 위치한다. Gt를 계산할 때 ifInt 함수는 >를 수행하여 False를 0으로, True를 1로 매핑하는 기능을 한다.

```
eval :: Prog -> Int
eval prog = head $ exec prog []
exec :: Prog -> Stack -> Stack
exec [] stck = stck          -- program execution terminated
exec (x:code)  stck = case x of
   Push n->    exec code (n:stck)
   Plus ->     let (s2:s1:ss) = stck
               in exec code ((s1+s2):ss)
   Mult ->     let (s2:s1:ss) = stck
               in exec code((s1*s2):ss)
   Gt ->       let (s2:s1:ss) = stck
               in exec code ((ifInt (>) s1 s2) : ss)
```

그림 8 순수 함수형 프로그래밍에 의한 산술 연산자 구현

이 프로그래밍에서 계산 과정은 스택의 변화를 통해서 이루어지며, 따라서 스택이 항상 exec의 파라메터로서 전달됨을 볼 수 있다. 스택 외에 다른 효과는 전혀 사용되지 않는 순수 함수형 프로그래밍 기법이 적용되고 있다.

3.2.2 모나드 방식에 의한 구현

eval 함수의 계산은 스택의 변환을 통해서 이루어지며, 그 결과 Int 타입의 값을 계산해 낸다. 이 상황에 상태 모나드를 적용하면, 어떤 한 명령어의 수행은 Stack -> (Int, Stack)로서 표현된다. 즉, 상태 모나드에서 상태 s가 스택으로 실례화 된다. 그림 9는 그림 8과 같은 의미의 프로그램으로서 모나드 방식을 이용하여 코딩한 것이다.

```
eval          :: Prog -> State Stack Int
eval []       = do {r <- pop ; return r}
eval (c: cs)  = do {exec c ; eval cs}
exec      :: Code -> State Stack ()
exec c =  case c of
   Push n -> do push n
   Plus ->   do {x <- pop; y <- pop; push (x+y)}
   Mult ->   do {x <- pop; y <- pop; push (x*y)}
   Gt ->     do {x <- pop; y <- pop; push $ ifInt (>) y}
```

그림 9 모나드 프로그래밍에 의한 산술 연산자 구현

그림 9에서 상태 모나드에 따라 스택은 하부구조에서 작동하며, 그 변화는 exec의 계산에 영향을 주지만 표면상으로 드러나지 않는다. 즉, exec이 스택을 효과(effect)로서 이용하면서 계산하는 프로그래밍을 적용하였다. exec은 한 명령어만을 처리하며, eval 함수가 재귀적으로 호출됨에 따라 exec 또한 반복 수행된다. exec의 수행 결과는 스택에 남아 있으므로 상태 모나드의 결과 값은 없다 (( )로 표현됨). eval은 프로그램 수행 종료시 스택에 남아 있는 결과 값을 pop하여 그 결과를 return 한다.

3.3 변수를 이용하는 연산자들의 구현

변수의 사용과 관련된 세 개의 명령어 Rvalue, Lvalue, Assgn이 이용된다. 명령형 프로그래밍에서 변수는 할당문의 왼쪽에 나타나는 경우와 그 외의 경우에 그 의미가 다르다. 할당문에 왼쪽에 나타나는 변수는 그 변수와 연관된 값을 담고 있는 공간의 주소를 의미하고 (Lvalue), 그 외에 나타는 변수는 그 공간에 있는 값 (Rvalue)을 의한다. Assgn은 어떤 한 Lvalue에 값을 할당하고 있음을 의미한다. 예를 들어, 두 할당문 x := 100; y := 3 + x + 2을 이 어셈블리 언어로 번역하는 과정은 다음과 같다. 먼저 이들을 후연산 표현 방식으로 표현하면 각각 "x 100 :=" 과 "y 3 x + 2 + :="으로 표현된다. 이들을 어셈블리 명령어로 번역한다면 [Lvalue "x", Push 100, Assgn, Lvalue "y", Push 3, Rvalue "x", Plus, Push 2, Plus, Assgn, Rvalue "y"]이 된다. 변수 값의 동적인 변화를 구현하는 수단으로 심볼테이블 Symtab을 이용하기로 한다.

3.3.1 순수 함수형 방식에 의한 구현

```
exec    :: Prog -> Symtab -> Stack -> Stack
exec ([],x) sym stck     = stck
exec (x:code) sym stck   = case x of
 Plus ->    let (s2:s1:ss) = stck
            in exec code sym ((s1+s2):ss
 Rvalue s-> exec code sym ((lkup s sym):stck)
 Lvalue s-> case index 0 s sym of
          0 -> exec code ((s, init):sym) (0:stck)
          i -> exec code sym (i:stck)
 Assgn ->  let (v:i:ss) = stck in exec code (update i v sym) ss
```

그림 10 순수 함수형 프로그래밍에 의한 변수 연산자 구현

어떤 변수에 할당된 값의 표현은 (variable, value)로 나타낼 수 있고, Symtab은 이들의 모임이다. Rvalue의 구현은 주어진 변수이름에 대한 value를 찾는 것으로서, 함수 lkup이 이 기능을 하고 있다. 찾아진 변수의 값은 스택에 적재된다. Lvalue를 처리할 때는 두 가지 경우를 고려해야 한다. 주어진 변수가 이미 Symtab에 등록되었을 수도 있고, 그렇지 않을 수도 있다. 여기서 index 함수는 심볼테이블 sym에 등록된 변수 s의 주소(리스트의 인덱스)를 알려 준다. index의 결과 값이 0이면 이 변수는 지금까지 심볼테이블에 등록된 적이 없음을 의미한다. 이 경우 심볼테이블의 맨 앞에 그 (이름, 초기값)을 등록시키며, 인덱스 값 0을 스택에 넣는다. 이미 등록된 변수에 대해서는 그 인덱스 값 i를 찾아서 스택에 넣는다. Assgn의 구현은 스택에 저장된 변수의 인덱스 주소 i와 이와 연관될 값 v를 가져와서, update 함수를 통해서 심볼테이블 sym을 업데이트함으로써 이루어진다.

3.3.2 모나드 방식에 의한 구현

그림 10의 Plus 연산자에 대한 정의는 그림 8의 경우와 달리 심볼테이블을 포함하도록 되어있다. Plus는 변수를 사용하지 않으므로 심볼테이블과 연관될 필요가 없음에도 불구하고 exec 함수를 이용하기 위해서는 이런 형태를 취할 수 밖에 없다. 이것은 순수 함수형 프로그래밍 형태로 코딩할 때 피할 수 없는 현상이다.

모나드 추상화를 이용하면 이 상황은 달라질 수 있다. 명령어들 중에서 Plus, Mult, Gt는 스택을 이용하여 계산하고, Rvalue, Lvalue, Assgn은 심볼테이블을 이용하여 계산한다. 이 두 명령어 그룹에 대해서 각각 두 상태 모나드(State Stack Int)와 (State Symtab Int)를 독립적으로 이용하여 계산할 수 있다. 이 상황에서 요구되는 것은 마치 함수를 합성하듯이, 이 두 모나드를 연동하여 사용할 수 있도록 합성시키는 것이 필요하다. 이를 위해서 모나드 트랜스포머가 이용된다. 다음 그림 11에서 StateT는 상태 모나드 트랜스포머를 의미한다.

```
ass    :: Prog -> StateT Stack (State Symtab) Int
exec   :: Code -> StateT Stack (State Symtab) ()
exec c  = case c of
   Plus ->        do { x <- pop; y <- pop; push (x+y)}
   Rvalue name->  do {v <- lkupUp name; push v}
   Lvalue name->  do i <- indexUp name
                     case i of
                       0 -> do { push i; pushUp (name, initNum) }
                       i -> push i
   Assgn ->       do { v <- pop; i <- pop; updateUp i}
```

그림 11 모나드 트랜스포머를 이용하는 변수 연산자 구현

3.3.3 상태 모나드 트랜스포머의 이용한 모나드의 합성

그림 11의 exec의 타입 정의에서, 두 개의 모나드 (State Stack Int)와 (State Symtab Int)를 합성하기 위하여, 이 둘 중에 하나를 트랜스포머로 정하고 나머지 하나를 내재시키는 방법을 이용한다. 여러 모나들을 트랜스포머로 계층을 형성할 때는 적재 순서에 따라 그 의미가 달라질 수 있음에 유의해야 한다. 여기에서는 주 연산이 스택을 위주로 일어나는 것을 고려하여 스택 상태 모나드를 위쪽에, 심볼테이블 상태 모나드를 아래쪽에 위치시키는 (StateT Stack (State Symtab) Int)의 형태가 되도록 하였다. 즉, 스택 상태 모나드가

트랜스포머가 된다.

그림 12와 같이 두 모나드를 연동할 때, 하부 모나드의 계산 결과는 lift를 이용하여 상부의 모나드로 전달될 수 있다.

```
lift   :: (MonadTrans t, Monad m) => m a -> t m a
lkup   :: Name -> State Symtab Value
index  :: Index -> Name -> State Symtab Index
update :: Index -> Value -> State Symtab ()
lkupUp      = lift . lkup
indexUp     = lift . index 0
pushUp      = lift . pushS
updateUp i  = lift . update i
```

그림 12 심볼테이블을 이용하는 함수 및 그들의 lifting

(StateT Stack (State Symtab) Int)에서, 하부의 (State Symtab) 모나드에 연관된 함수 lkup, index, update의 계산 결과를 상부의 (StateT Stack) 모나드로 전달하기 위해 이들을 lift 함수와 합성시켜, 각각 lkupUp, indexUp, updateUp 함수가 정의되었다. lift의 타입은 (m a -> t m a)로서, 한 트랜스포머 t와 모나드 m이 정의되었을 때, m의 계산 결과를 트랜스포머 t로 전달하는 기능을 한다.

3.4 Goto 및 제어 흐름 명령어들의 구현

제어 흐름을 위한 세 명령어 (Lab label), (Goto label), (GoFalse label)의 구현에 대해서 논의한다. Lab은 프로그램의 어떤 한 지점에 라벨 n을 지정하는 역할을 하며 그 밖의 기능은 없다. 즉, 스택이나 심볼테이블에서의 변화는 일어나지 않는다. (Goto label)은 프로그램 중에서 (Lab label) 명령어가 위치한 그 다음 명령어로 제어를 넘긴다. 이 명령어의 수행 또한 스택이나 심볼테이블에 아무런 영향을 주지 않는다. GoFalse label은 스택의 탑에 위치한 값을 pop 하여, 그것이 0 인 경우 (Goto label)을 수행한다.

이 명령어들 구현의 쉬운 한가지 방법으로서, 그림 13과 같이 이미 수행된 프로그램을 저장하는 방법을 이용한다. 수행될 프로그램이 저장된 코드가 (c:cs)의 형태로 되어 있을 때, c 코드를 수행한 후 이를 버리지 않고 저장하는 것이 필요하다. 이를 위해서 프로그램의 수행은 Prog에서 튜플 (Prog, Save)로 확장된다. 프로그램 ((c:cs), save)을 명령어 c의 exec 수행 후 (cs, (c:save))로 변화시켜, 수행된 명령어가 저장되도록 한다. 이 튜플 (c, s)는 언제나 원래의 프로그램을 회복할 수 있다. 즉, Prog = (reverse s) ++ c 의 법칙이 성립한다. Goto label의 구현은 goto 함수에 의해서 처리되는데, 이 함수는 (c, s)로부터 원래의 프로그램 org를 회복한 후, org에서 (Lab label)이 위치한 곳을 기준으로 org를 두 개로 분리하여 새로운 (c', s')을 만들어 내면 된다. 위에 기술한 동작의미를 바탕으로 ass를 다음 그림 13과 같이 수정한다.

4. 모나드에 의한 프로그램 구성의 모듈화

[10,11]은 모나드/모나드 트랜스포머 기법을 이용함으로써 모듈성을 갖는 프로그램 개발이 가능함을 제시하였다. 여기에서는 그들이 제시한 원리에 따라, 본 연구에서 개발한 프로그램 코드가 어떤 형태의 모듈성을 갖는지를 분석해 본다.

Goto 및 제어 흐름 기능을 포함하는 어셈블리 언어 인터프리터를 순수 함수형 프로그래밍 방식으로 코딩하면 아래 그림 14와 같다. 이 프로그래밍에서는 지금까지 논의되었던 스택, 심볼테이블, 코드 저장 등의 기능이 모두 적용되었다. exec 함수의 타입을 그림 8과 그림 14의 두 경우를 비교하면 다음과 같다.

```
exec :: Prog -> Stack -> Stack
exec :: ([Code],[Code]) -> Symtab -> Stack
-> Stack
```

기능이 추가되면서 exec 함수 또한 더욱 복잡해지는 것을 알 수 있다. 그림 8과 그림 14에서 exec이 Plus 명령어를 구현하는 상황을 비교하면 다음과 같다.

```
Plus -> let (s2:s1:ss) = stck in exec code
((s1+s2):ss)
```

```
ass :: (Prog, Save) -> StateT Stack (State Symtab) Int
ass ([],_)          = do {r <- pop ; return r}
ass ((c:cs),save)  = case c of
  Lab n ->  do  ass (cs, (c:save))
  Goto lab -> do  ass $ goto ((c:cs), save) lab
  GoFalse lab -> do x <- pop
                    if x == 0 then ass $ goto ((c:cs), save) lab
                              else ass (cs,(c:save))
goto :: (Prog, Save) -> Int -> (Prog, Save)
goto (code,sav) lab  = let org = (reverse sav) ++ code
                           (sav',(labIns:code')) = span (/= (Lab lab)) org
                       in (code', labIns:(reverse sav'))
```

그림 13 Goto 및 제어 흐름 명령어들의 구현

```
exec :: ([Code],[Code]) -> Symtab -> Stack -> Stack
exec ([],x) sym stck = stck
exec ((x:code),sav) sym stck = case x of
  Plus ->      let (s2:s1:ss) = stck in exec (code,x:sav) sym ((s1+s2):ss)
  Rvalue s -> exec (code,x:sav) sym ((lkUp s sym):stck)
  Lvalue s -> case index 0 s sym of
                0 -> exec (code,x:sav) ((s, init):sym) (0:stck)
                i -> exec (code,x:sav) sym (i:stck)
  Assgn ->     let (s2:s1:ss) = stck  in exec (code,x:sav) (update s1 s2 sym) ss
  Plus ->      let (s2:s1:ss) = stck  in exec (code,x:sav) sym ((s1+s2):ss)
  Gt ->        let (s2:s1:ss) = stck in exec (code,x:sav) sym (ifInt (>) s1 s2:ss)
  Lab n ->     exec (code,x:sav) sym stck
  Goto lab -> exec (goto (x:code) sav lab) sym stck
  GoFalse lab -> let (s:ss) = stck
                 in if s==0 then exec (goto (x:code) sav lab) sym ss
                            else exec (code,x:sav) sym stck
```

그림 14 코드 저장, 심볼테이블 및 스택을 이용하는 순수 함수형 프로그래밍

```
Plus -> let (s2:s1:ss) = stck in exec
(code, x:sav) sym ((s1+s2):ss)
```

Plus의 구현은 심볼테이블의 구현과 상관이 없지만, 이 코드에서 볼 수 있듯이 exec이 변수 처리를 위해 심볼테이블의 사용을 도입함에 따라 Plus에 대한 코드 또한 심볼테이블과 관련된 부분을 포함하도록 확장되었다. 즉, exec의 기능 추가가 필요한 곳만 정확하게 적용되는 것이 아니라 exec 함수 전체에 적용되고 있음을 볼 수 있다. 이상적으로, 어떤 한 기능이 추가될 때, 정확하게 관련된 곳만을 수정하고 나머지는 원래의 모습을 유지하는 원리를 갖는 것이 바람직하지만, 이 구현에서는 그 원리가 적용되지 않고 있다.

이 현상은 순수 함수형 프로그래밍이 프로그램 모듈화를 제대로 표현하지 못하고 있음을 보여주는 한 예라고 보여진다. 프로그램 부분(fragments)의 합성 과정을 반복하여 전체 프로그램을 구성하는 데 있어서, 위의 예는 프로그램의 합성이 기존 프로그램의 수정을 요구하지 않는 것이 바람직하지만, 순수 함수형 프로그래밍에서는 이 원리가 적용되지 않는다. 이 문제는 명시적 의미론(denotational semantics)의 모듈화 부재의 논의와 연관된다[11].

순수 함수형 프로그래밍의 모듈성 부재의 문제는 모나드 트랜스포머를 사용함으로써 해결될 수 있음이 제시되었다[10,11]. 모나드 추상화는 효과를 이용하는 프로그래밍을 가능케 하며, 응용 분야의 특성을 최적으로 표현하기 위한 프로그래밍 환경을 제공해 준다. 각 분야의 특성에 맞게 개발된 모듈들은 모나드 트랜스포머에 의해 합성될 수 있다. 이런 개념은 일반적으로 다양한 분야에 적용될 수 있으며, 본 연구도 이 원리를 따르고 있음을 볼 수 있다.

모나드를 이용하여 프로그래밍하는 경우, Plus를 위한 exec의 구현은 exec에 기능이 추가됨에 상관없이 그림 11과 그림 9에서 모두 동일함을 볼 수 있다.

5. 관련 연구 및 결론

하스켈은 순수 함수형 언어지만 모나드를 이용함으로써 명령형 프로그래밍처럼 효과를 이용하는 스타일의 프로그래밍도 가능하다. 모나드는 일종의 추상화 기법으로 볼 수 있는데, 이 추상화를 이용함으로써 상위부분에서 핵심 부분을 기술하고, 하부 구조에서 기반 환경을 구현하도록 시스템을 구성할 수 있다. 이러한 계층적 구조는 상부에 있는 프로그램을 더욱 간결하게 보일 수 있도록 한다. 또한, 각 응용 분야의 특성에 그에 적합한 모나드를 정의하고 이들을 통합함으로써, 통일적이고 일관성있는 개념으로 프로그램을 구성할 수 있다.

모나드 기법을 이용한 인터프리터 개발은 [12,13]에서 시작되었으며, 이 기법은 [10]에서 트랜스포머에 의한 모듈성을 갖는 프로그래밍 언어 인터프리터의 개발로 발전되었다. [10]에서는 상태 모나드 외에 예외처리, Reader 등의 다양한 모나드 기법이 포괄적으로 소개되고 있으며, 대상 프로그래밍 언어는 함수의 적용, callCC, 비결정적 계산 등의 기능을 갖는 고급 프로그래밍 언어이고, 인터프리터의 구현은 Gofer를 이용하였다. 이와 비교할 때, 본 연구는 어셈블리 언어를 기반으로 하고 있으며, 상태 모나드와 상태 모나드 트랜스포머를 이용하는 구체적인 프로그래밍 방법을 소개하고 있다.

프로그램을 구성하는 데 있어서, 모나드 프로그래밍을 이용하여 각 부분별 특성에 가장 적합한 프로그램 부분들을 개발하고 이들을 합성함으로써 프로그램을 구성하는 개념은 더욱 일반화되어 발전되고 특수 목적용 언어(Domain Specification Programming Languages) 분야로 진화되고 있다[14].

본 연구에서는 스택 기반의 어셈블리 언어의 인터프리터를 모나드 추상화 기법으로 표현하였다. 순수 함수형 프로그래밍 기법과 비교하였을 때, 그 추상화는 매우 간결하고 명확하게 표현됨을 볼 수 있다.

보통 하스켈 프로그래밍을 배울 때 모나드와 모나드 트랜스포머에 대해서 어려움을 느끼는 경우가 많다. 상태 모나드와 상태 트랜스포머는 여러 모나드 중에서 가장 핵심적인 모나드로서, 본 연구에서 소개하는 순수 함수형 프로그래밍 및 그와 대조되는 상태 및 상태 모나드 트랜스포머를 이용하는 프로그래밍 기법은 하스켈 학습자들에게 도움이 될 수 있을 것으로 사료된다.

참 고 문 헌

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

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

[3] Graham Hutton. *Programming in Haskell*, Cambridge University Press, 2007.

[4] Simon Thompson. *Haskell: The Craft of Functional Programming*, Addison Wesley, 1999.

[5] Haskell Homepage. http://haskell.org.

[6] MTL (Monad Transformer Library). Control.Moad.

[7] Bryan O'Sullivan, Don Stewart, and John Goerzen, *Real World Haskell*, O'Reilly Media, 2008.

[8] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*, pp.97-136, 1995.

[9] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Pearson Education, 1986.

[10] Sheng Liang, Paul Hudak, and Mark P. Jones, Monad transformers and modular interpreters. In *POPL*, pp.333-343, 1995.

[11] Sheng Liang, *Modular Monadic Semantics and Compilation*, PhD thesis, Yale University, 1997.

[12] David Espinosa, Building interpreters by transforming stratified monads, 1994.

[13] Guy L. Steele Jr, Building interpreters by composing monads, In *POPL* '94, January 1994.

[14] Paul Hudak, Modular Domain Specific Languages and Tools, Proceedings of Fifth International Conference on *Software Reuse*, IEEE Computer Society Press, 1998.

변 석 우

1976년~1980년 숭실대학교 전자계산(학사). 1980년~1982년 숭실대학교 전자계산(석사). 1982년~1999년 ETRI 책임연구원. 1988년~1994년 영국 University of East Anglia 전산학(박사). 1998년~현재 경성대학교 컴퓨터학부 교수. 관심분야는 함수형 프로그래밍, 정형 증명, 프로그래밍 언어 등