애로우를 이용한 오류 처리 기법

이동주°, 지정훈, 장한일, 우균
부산대학교 컴퓨터공학과
{mrlee°, jhji, daystar, woogyun}@pusan.ac.kr

Error Handling Technique Using Arrows

Dongju Lee°, JungHoon Ji, Hanil Jang, Gyun Woo
Dept. of Computer Engineering, Pusan National University

요 약

모나드의 일반화 버전인 애로우(Arrow)는 모나드에 비해 효율적이며, 프로그램 합성을 위한 직관적인 인터페이스를 제공한다. 여러 프로그램을 합성할 때 합성된 전체 프로그램의 오류 처리는 매우 중요한 문제이다. 각각의 프로그램에서 오류 처리를 일일이 기술하는 것은 매우 번거로우며 비효율적인 작업이기 때문이다. 본 논문에서는 애로우 인터페이스를 이용하는 프로그램을 합성할 때 효율적으로 오류를 처리하기 위한 방법을 제시한다. 모든 애로우 타입에 대하여 오류를 처리하기 위해 새로운 애로우를 정의하며 이를 프로그램 합성 단위로 이용하여 전체적인 프로그램의 오류를 처리한다. 또한 애로우를 이용한 타입 검사 프로그램을 통하여, 논문에서 제시한 오류 처리 기법을 적용하여 효율성을 평가한다.

1. 서 론

순수 함수형 언어인 하스켈(Haskell)은 명령형 언어에서 제공하는 문장(statement) 및 대입 연산자(assignment operator)를 제공하지 않는다[1]. 따라서 순차적인 수행관계를 갖는 프로그램, 상태를 이용하는 프로그램, 특 입출력 프로그램을 표현하는데 어려움이 있을 것이라고 생각하기 쉽다. 하지만 이와 같은 문제점은 모나드(Monad)를 통하여 해결되었다[3].

모나드는 상태 또는 순차적인 계산 과정을 추상화하는데 사용될 뿐만 아니라 프로그램 조각을 합성하는 기법으로도 이용된다[2]. 프로그램 조각을 라이브러리로 볼 경우, 모나드는 정형화된 인터페이스를 제공하는 컴비네이터(combinator) 라이브러리로 볼 수 있다. 컴비네이터는 프로그램을 합성할 때 이용되며, 프로그램을 입출력으로 다루는 함수이다. 이와 같이 함수의 입력이나 출력으로 프로그램을 다룰 수 있는 것은 순수 함수형 언어에서 함수를 일등급값(first-class value)으로 다루기 때문이다[1].

애로우(Arrow)는 모나드를 일반화한 새로운 계산 인터페이스로 2000년 John Hughes에 의해 발표되었다[4]. 애로우는 모나드처럼 프로그래밍 합성을 위해 일반화된 인터페이스이다. 모나드와 차이점은 계산의 결과 뿐 아니라 계산의 입력에 대해서도 추상화를 제공한다는 점이다. 따라서 입출력이 결정된 애로우 기반의 프로그램 조각은 단지 컴비네이터만을 이용하여 합성이 가능하기 때문에 기존의 모나드에 비하여 쉽게 프로그램을 합성할 수 있다.

프로그램 합성 시 각각의 프로그램에 대한 오류 처리는 주요한 문제 중 하나이다. 조각 프로그램의 내부 오류가 생길 경우, 원하지 않는 조각 프로그램의 출력 값이 생성될 경우 전체 프로그램이 계산은 실패하게 된다. 이와 같이 프로그램을 합성할 때 각 프로그램의 오류 처리를 일일이 기술하는 것은 매우 번거로우며 비효율적인 작업이다. 본 논문에서는 애로우를 이용한 효율적인 오류 처리 방법을 제시하고자 한다.

본 논문의 구성은 다음과 같다. 2장에서는 애로우에 기본적인 개념과 하스켈에서 정의된 애로우 클래스에 대해서 살펴본다. 3장에서는 오류처리를 위한 Error 펑터(functor)를 정의하며, 4장에서는 오류 처리 기법이 적용된 타입검사 프로그램을 작성한다. 끝으로 작성된 프로그램을 중심으로 오류 처리 기법의 효율성을 평가하고 향후 발전 방향에 대해서 논한다.

2. Arrow

애로우는 그림 1에서 나타낸 바와 같이 입력과 출력을 가지는 계산을 추상화한 것이다. 따라서 애로우 타입은 입출력에 대한 2개 이상의 타입 파라미터를 가지는 고차타입으로 구성된다.

그림 1. 애로우의 계산 모델

Hughes는 하스켈에서 애로우를 이용하기 위해 Arrow 클래스를 정의하고 있다[4].

```
class Arrow a where
    arr :: (b -> c) -> a b c
    (>>>) :: a b c -> a c d -> a b d
    first :: a b c -> a (b,d) (c,d)
```

그림 2. Arrow 클래스

Arrow 클래스는 3가지 컴비네이터로 구성된다. arr는 b 타입의 입력과 c 타입의 출력을 가지는 함수를 애로우로 전환하는 함수이다. (>>>)는 순차적인 합성 연산자라고 부르며, 첫 번째 애로우의 계산 결과를 두 번째 애로우의 입력으로 연결하며, 두 애로우를 하나의 애로우로 합성하는 역할을 한다. 마지막으로 first는 입력 튜플(tuple)의 일부분

만 계산에 적용하도록 애로우를 확장한다. first는 주로 다른 컴비네이터를 정의할 때 이용된다. 다음 그림 3은 프로그램 합성 시 자주 이용되는 애로우 컴비네이터들이다.

```
second  :: Arrow a => a b c -> a (d, b) (d, c)
(***)   :: Arrow a =>
           a b c -> a b' c' -> a (b, c') (c, c')
(&&&)   :: Arrow a =>
           a b c -> a b c' -> a b (c, c')
(|||)   :: ArrowChoice a =>
           a b d -> a c d -> a (Either b c) d
```

그림 3. 자주 이용되는 애로우 컴비네이터들

현재 애로우는 Haskell98 표준라이브러리[1]로는 제공되지 않고 있지만, GHC 6.4.1버전에서는 Control.Arrow라는 모듈을 통해 제공하고 있다.

3. Error Functor

Hughes의 논문에서는 모든 애로우 타입에 대해 오류처리를 지원하는 Maybe 펑터를 정의하고 있다[4]. 하지만 Maybe 펑터는 계산의 성공 또는 실패여부에 대해서만 처리하기 때문에, 여러 프로그램을 합성할 때 어느 부분에서 오류가 발생했는지를 쉽게 판단할 수 없다. 따라서 본 논문에서는 Maybe 펑터를 확장한 Error 펑터를 정의한다.

다음 그림 4는 ErrorFunctor의 타입과 애로우 인터페이스의 인스턴스(instance)이다. ErrorFunctor 타입은 타입 파라미터로 a, b, c를 갖는데, 이 때 (a b c)는 ErrorFunctor가 포함하는 다른 애로우 타입을 나타낸다. ErrorFunctor 타입의 정의 부분을 살펴보면, a 타입 애로우의 출력 타입c를 Either c ErrorMsg 타입으로 변형한다. 따라서 ErrorFunctor를 이용할 경우 입력에 의해 계산 결과가 올바를 경우 Left c 값을 가지며, 오류가 있을 경우 Right ErrorMsg값을 가진다. 이 때 ErrorMsg는 오류 메시지를 나타낸다.

liftErrorF 함수는 Arrow 클래스로 정의된 어떤 애로우를 ErrorFunctor 애로우로 전환한다. errMsg 함수는 출력 타입으로 Maybe 타입을 가지는 애로우에 오류 메시지를 포함하여 ErrorFunctor 애로우로 전환한다. 이 함수는 두 번째 파라미터로 emsg를 받는데, 이는 계산 결과가 Nothing일 때 ErrorFunctor의 오류 메시지로 이용된다. 여기서 생성자 EF 안에 있는 arr는 ErrorFunctor 애로우의 arr 함수로 착각할 수 있는데, 이는 ErrorFunctor에 포함된 애로우 a의 arr 함수이다.

ErrorFunctor의 arr 함수는 liftErrorF 함수를 이용하여 정의된다. arr의 정의 부분에 있는 arr는 ErrorFunctor가 포함하는 애로우의 arr이다. 따라서 ErrorFunctor의 arr는 일반적인 함수를 ErrorFunctor가 포함하는 애로우로 생성한 다음 이를 다시 ErrorFunctor로 변환하는 두 가지 과정을 거치게 된다. 두 ErrorFunctor 애로우를 합성하는 (>>>)는 첫 번째 애로우의 결과 값이 Left 일 때 그 값을 두 번째에 전달하며, Right 일 경우 첫 번째 오류메시지를 포함하는 Right를 전체 프로그램의 결과 값으로 취급한다.

4. 타입 검사 애로우

본 논문에서는 RT-FRP[5]의 표현(expression)에 대한 타입 검사 프로그램을 애로우 기반으로 작성하였다. 그림 5는 표현의 데이터 타입과 정수 상수, Let 구문의 타입 검사 프로그램이다. RT-FRP의 표현은 ExpTerm 데이터 타입으로 구성되며, 각 ExpTerm의 타입은 TermType의 값에 해당된다. 각각의 ExpTerm에 대한 타입 검사 프로그램은 Trace 애로우로 구성된다. Trace 애로우는 변수에 대한 타입정보를 가지는 타입 환경(type environment)을 입력으로 받고 TermType 타입을 출력 값으로 낸다.

그림 5에서는 ExpTerm중 상수와 Let 구문에 대해서만 타입 검사 애로우를 정의하였다. 정수 상수의 타입은 항상 TInt 값을 갖는다. 따라서 입력에 관계없이 TInt 값을 반하는 함수를 이용하여 Trace 애로우를 구성한다.

```
type ErrorMsg = String
newtype ErrorFunctor a b c = EF (a b (Either c ErrorMsg))

liftErrorF :: Arrow a => a b c -> ErrorFunctor a b c
liftErrorF f = EF (f >>> arr Left)

errMsg :: (Arrow a) => a b (Maybe c) -> ErrorMsg -> ErrorFunctor a b c
errMsg f emsg = EF (f >>> arr (\a -> case a of Just k -> Left k
                                               Nothing -> Right emsg))
instance ArrowChoice a => Arrow (ErrorFunctor a) where
    arr f          = liftErrorF (arr f)
    EF f >>> EF g  = EF (f >>> (g ||| arr Right))
    first (EF f)   = EF (first f >>> arr(\(c',d) -> case c' of Left c -> Left (c,d)
                                                               Right e -> Right e))
```

그림 4. ErrorFunctor 타입 정의 및 Arrow 클래스 인스턴스 정의

```
type Env a = [(NAME,a)]
data ExpTerm
    = Let   NAME ExpTerm ExpTerm
    | BinOp Op ExpTerm ExpTerm
    | ConI  Int
    | Var   NAME
    ....
data TermType = TInt | TBool
        ....
expType1 :: ExpTerm
    -> Trace (Env TermType) TermType
expType1 (ConI I) = arr (\_ -> TInt)
expType1 (Let name e1 e2)
        = ((expType1 e1) &&& (arr id))
          >>> arr (\(t1,env) -> (name,t1):env)
          >>> expType1 e2
```

그림 5. 표현의 데이터 타입 및 타입 검사 애로우

Let 구문의 타입 계산 규칙에서 기호 name의 타입은 e1 타입과 같고, name을 이용하는 e2의 타입이 전체 Let 구문의 타입이다. 따라서 Let 구문은 e1 타입을 계산하는 애로우와 타입 환경에 기호를 추가하는 애로우, e2 타입을 계산하는 애로우의 순차적인 합성으로 전체 프로그램을 구성할 수 있다.

이항 연산자(binary operator)를 이용하는 구문과 변수(variable)를 나타내는 구문에 대해서 살펴보자. 대부분의 이항 연산자는 피연산자의 타입이 정해져 있다. 예를 들어 논리 연산자인 & 일 경우 오퍼랜드의 타입 값으로 TBool만 가능하다. 이와 같은 규칙이 성립되지 않을 경우, 현재 타입 검사는 실패한 것으로 간주되어야 한다. 또한 첫 번째 피연산자 구문의 타입 검사에서 오류가 발생할 경우 두 번째 피연산자 구문을 살펴볼 필요도 없이 전체 구문의 타입 검사는 실패한 것으로 간주해야한다.

변수의 기호를 나타내는 Var 구문도 마찬가지다. 타입 환경에서 기호가 없을 경우 이 구문에 대한 타입 계산은 실패한 것이다. 이와 같이 대부분의 프로그램은 계산 실패에 대한 오류 처리가 필수적이며, 이는 특 프로그램 합성 시 매우 중요한 역할을 한다.

다음 그림 6은 앞서 3장에서 정의한 Error 펑터를 기반으로 오류 처리를 추가한 타입 검사 프로그램이다. Var 구문에서는 입력으로 받는 타입 환경에서 기호 name이 있는지를 lookup 함수를 이용하여 검사한다. lookup 함수는 찾는 키가 있을 경우 Just, 없을 경우 Nothing을 반환한다. Trace의 arr을 이용하여 Trace 애로우를 하나 정의한 다음, errMsg 함수를 이용하여 ErrorFunctor 애로우로 전환한다. 마찬가지로 BinOp 구문도 오류가 발생할 여지를 가지는 부분에 대해서 errMsg 함수를 이용하여 ErrorFunctor로 구성하면 된다.

ErrorFunctor를 이용할 경우 모든 애로우를 대상으로 쉽고 간결하게 오류기능을 추가할 수 있다. 그림 5의 expType1도 타입 선언만 expType2와 동일하게 바꾸면

```
expType2 :: ExpTerm ->
 ErrorFunctor    Trace    (Env    TermType)
TermType
expType2 (Var name)
    = arr (\env -> lookup name env) `errMsg`
        ("Variable"++ name ++"is not defined")
expType2 (BinOp op e1 e2)
    = ((expType2 e1) &&& (expType2 e2))
      >>> (arr (\(t1,t2) -> bopType op t1 t2))
            `errMsg`
            ("Operand" ++ (show e1) ++ "and"
            ++ (show e2) ++ "type not valid")
```

그림 6. 오류 처리 기능이 추가된 타입 검사 애로우

오류기능이 추가된 프로그램이 된다. 이와 같은 특징은 미리 정의된 ErrorFunctor의 컴비네이터와 errMsg 함수가 대부분의 오류처리를 다루며 합성된 프로그램에는 이들 컴비네이터에 오류처리 메커니즘이 숨겨져 있기 때문이다. 만약 ErrorFunctor를 사용하지 않고 오류 처리를 할 경우, 프로그램을 합성할 때 각각 프로그램의 성공, 실패와 메시지를 전달하는 애로우를 일일이 작성한 뒤 합성하야 할 것이며, 이는 프로그램을 매우 복잡하게 하는 요소가 될 것이다.

5. 결론

애로우 인터페이스는 프로그램의 입력과 출력을 블록 형태로 추상화한 것으로 볼 수 있다. 따라서 애로우 인터페이스로 추상화된 프로그램 조각은 입력과 출력을 연결하는 것만으로 합성할 수 있다. 즉 애로우를 이용하면 블록을 쌓듯이 프로그램을 합성할 수 있다.

본 논문에서는 애로우 기반의 프로그램 합성 시 효율적인 오류 처리를 위한 프로그램 합성 단위인 Error 펑터를 정의하였다. 또한 이를 기반으로 오류 처리가 적용된 타입 검사 프로그램을 작성하였다. Error 펑터를 이용할 경우 비교적 쉽고, 간결하게 프로그램을 합성할 수 있다는 것을 RT-FRP의 타입검사 프로그램을 통해 보였다.

참고문헌

[1] S. P. Jones and et al, eds., *Ha kell 98 Language ans Librarie , the Revi es Report.* CUP, Apr. 2003.

[2] P. Wadler, "The essence of functional programming," in *POPL*, pp. 1-14, 1992.

[3] S. P. Jones, "Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell," *Engineering theorie of oftware con truction*, pp47-96, 2001.

[4] J. Hughes, "Generalising monads to arrows," *Science of Computer Programming*, vol. 37, pp. 67-111, May 2000.

[5] Z. Wan, *Functional Reactive Programming for Real-Time Reactive Sy tem* . Ph.D. dissertation, Computer Science Department, Yale University, Oct. 2002.