프로그램 합성 관점에서 지연 함수형 언어의 예외처리 기법

이동주° 우균
부산대학교 컴퓨터공학과
{mrlee°, woogyun}@pusan.ac.kr

Exception Handling Technique in Lazy Functional Language from the Viewpoint of Program Synthesis

Dongju Lee° Gyun Woo
Dept. of Computer Engineering, Pusan National University

요 약

순수 함수형 언어에서 예외처리를 구현하는 것은 매우 까다로운 문제이다. 지연계산, 참조투명성과 같은 주요 특징은 예외 처리와 상반된 성질을 가지는 때문이다. 예외의 처리순서는 계산순서와 관계가 있고, 예외의 발생순서는 참조투명성과 밀접한 관계가 있다. 본 논문은 현재 하스켈(Haskell)에서 구현된 예외처리 방법의 분석을 통해, 프로그램 수행 시 효율적인 예외처리 방법에 대해서 제시한다. 합성된 프로그램에서 예외 발생할 때 예외가 전달되는 것을 사전에 차단하는 방법을 이용한다. 실제 예외가 발생한 프로그램을 작성하고, 프로파일링을 통하여 이 방법의 효율성을 점검한다.

1. 서 론

예외처리는 견고한 프로그램(robust program)을 만들기 위한 필수적인 요소로서 최근에 개발된 프로그래밍 언어에서 기본적으로 제공되는 기능이다. 예외처리는 잘못된 수행 상황에 대처할 수 있도록 프로그래머에게 표준적인 인터페이스를 제공한다. 대표적으로 명령형 언어인 C++, 자바(Java)에서는 try-catch 구문을 제공하고 있다.

프로그램 수행 시 예외의 발생은 순서가 있다는 특징이 있다. 즉 동일하게 발생된 예외는 존재하지 않으며, 예외처리 또한 발생된 순서대로 진행된다. 계산의 순서를 가지는 명령형 언어는 비교적 자연스럽게 예외처리가 구현되지만, 계산의 순서가 없는 지연 함수형 언어에서는 매우 까다로운 문제이다[3].

순수 함수형 언어인 하스켈(Haskell)의 특징 중 지연계산(lazy evaluation)과 참조투명성(referential transparency)은 예외처리의 구현을 힘들게 하는 요소이다. 지연계산은 예외의 발생 순서를 결정하기 힘들게 하며, 만약 계산순서가 결정되어 예외의 발생순서가 정해지더라도 계산 순서에 따라 각각 다른 예외를 발생시키므로 참조투명성에 어긋나게 된다. 이 문제를 해결하기 위해 현재 하스켈은 예외를 집합으로 구성하며, 발생된 예외는 IO 모나드(IO monad)에서 처리하도록 하고 있다[5].

본 논문은 여러 함수가 합성된 함수의 일부분에서 예외가 발생할 때 예외가 전달(propagation)되는 현상을 구체적으로 살펴본 뒤, 문제점을 파악한 다음 효율적인 예외처리를 위한 방법을 제시한다.

본 논문의 구성은 다음과 같다. 2절에서는 지연 함수형 언어에서 예외처리 구현시 알려진 문제점과 현재 구현되어진 방법에 대해서 살펴본다. 3절에서는 합성된 프로그램에서 예외가 전달되는 상황을 살펴본 다음, 효율적으로 예외를 처리하는 방법을 모색한다. 4절에서는 연구결과를 분석하여 논문에서 제시한 방법의 적합성을 검토한다. 끝으로 결론을 내리며 향후 발전 방향에 대해서 논한다.

2. 지연 함수형 언어에서의 예외처리

지연 함수형 언어는 예외를 값으로 취급한다. 순수 값(purely value)은 정상 값(ordinary value) 또는 예외 값(exceptional value)으로 구성된다.

```
data ExVal a = Ok a | Bad Exception
throw    :: Exception -> a
catch    :: a -> ExVal a
f x = case catch (test x) of
          OK result -> result
          Bad ex -> recover_goop a
```

그림 1 예외 값을 처리하는 프로그램

그림 1에서 throw는 예외 값을 만드는 함수이다. 예외를 발생시키는 함수이다. catch는 값이 예외 값인지 아닌지를 판단하여, 고차 타입인 ExVal의 값을 낸다. 함수 f는 test 함수에 대해 예외 처리를 추가한 함수이다. 위 프로그램은 별 문제 없이 보인다. 하지만 test 함수가 그림 2와 같이 정의될 경우를 생각해보자.

```
test x = (throw ex1) + (throw ex2)
```

그림 2 예외선택 문제

그림 2의 test 함수는 두 가지 예외를 포함하고 있다. 하지만 어떤 예외를 먼저 발생시켜야 될지는 알 수 없다. 지연 함수형 언어에서는 정해진 계산 순서가 없기 때문이다. 만약 계산순서가 있다고 가정하고 하자.

```
ε (throw ex1) + (throw ex2)
        = ε (throw ex2) + (throw ex1)
```

그림 3 참조투명성 문제

그림 3의 두 표현은 의미(semantic)상 동일한 표현이다. 하지만 계산순서에 따라 발생되는 예외가 다를 경우, 다른 값을 가지게 되며 위 두 표현은 다른 의미를 가지게 된다. 이 경우, 계산순서에 무관하게 입력에 대해 항상 동일한 출력을 유지하는 참조투명성의 원리가 성립되지 않는다. 위 두 가지 문제를 통틀어 '+ problem'이라 부르며, 지연 함수형 언어에서 잘 알려진 문제이다[4].

하스켈은 이 문제를 해결 위해 예외를 집합으로 정의한다. 그림 3과 같이 동시에 발생하는 예외의 경우, 두 예외 값 모두를 포함하는 예외를 발생시킨다. '+'연산자의 의미가 그림 4와 같이 정의 될 경우, '+ problem'의 참조투명성 문제는 쉽게 해결된다. 그림 4에서 +'는 의미영역(semantic domain)에서 이용되는 연산자이다.

```
ε e1 + e1  = ε e1  +' ε e2
 where
 (Ok v1) +' (Ok v2) = Ok (v1+v2)
 (Ok v1) +' (Bad s2) = Bad s2
 (Bad s1) +' (Ok v2) = Bad s1
 (Bad s1) +' (Bad s2) = Bad (s1 ∪ s2)
```

그림 4 '+' 연산자의 의미

예외를 예외 값의 집합으로 구성할 경우, 예외를 처리하는 부분에서 어떤 예외를 선택할지도 중요한 문제이다. 현재 하스켈은 IO 모나드에서 발생한 예외를 결정한다.

```
evaluate :: a -> IO a
throw :: Exception -> a
catch :: IO a -> (Exception -> IO a) -> IO a
```

그림 5 GHC의 예외처리 함수

그림 5의 evaluate 함수는 타입 a의 값이 예외인지 아닌지를 점검한다. 예외가 아닐 경우, IO 모나드로 전환한다. 예외일 경우, 예외 값 중 하나를 선택하여 IO 예외(IO Exception)를 발생시킨다. throw는 예외를 발생시키는 함수이며, catch는 IO예외에 대해서 처리하는 함수이다. 그림 5의 예외처리 라이브러리는 표준 Haskell98 라이브러리로 제공되고 있지는 않지만, GHC 기본 패키지의 Control.Exception 라이브러리로 제공되고 있다.

3. 프로그램 합성 관점에서 효율적인 예외처리 방법

함수형 프로그램은 여러 함수가 합성된 형태로 구성된다. 함수를 합성하는 방법은 함수적용(function application)과 같은 아주 기본적인 방법부터 모나드나 애로우(Arrow)와 같은 정형화된 컴비네이터(combinator) 라이브러리를 이용하는 방법이 있다[2,6].

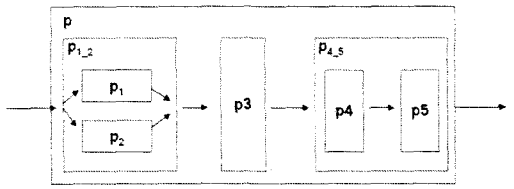

그림 6 프로그램 합성

그림 6은 합성된 프로그램을 나타내는 그림이다. 프로그램 p는 조각 프로그램 $p_{1..5}$가 합성된 프로그램이다. p_3에서 예외가 발생했다고 가정하자. 이때 가장 적합한 예외처리 방법은 예외가 $p_{4..5}$ 프로그램에게 전달되지 않고, 즉시 p_3을 구성하는 p에게 예외를 전달하는 것이다. 합성된 프로그램에서는 예외 발생 즉시, 예외를 처리 할 수 있도록 처리기(handler)에게 기회를 주는 것이 가장 효율적인 방법으로 볼 수 있다. 다음 그림 7은 조각 프로그램에서 예외가 발생하더라도 예외가 다른 조각 프로그램에게 전달되는 프로그램의 예이다.

```
fun1 :: Int -> (Int,Int)
fun1 n = (n, n')
 where n'
        | n<0 = throw (assert "neg value")
        | n == 0 = 0
        | n>20 = throw (assert "big number")
        | otherwise = n-1

fun2 :: (Int, Int) -> Int
fun2 (p1,p2) = (nfib1 p1) + (nfib2 p2)
fun3 = fun2 . fun1

test1 :: Int -> (IO Int)
test1 n = catch (evaluate (fun3 n)) (\e -> return 0)
main = do out <- test1 21
          print out
```

그림 7 예외가 전달되는 프로그램의 예

fun3은 fun1과 fun2를 합성함수 연산자를 이용하여 합성한 함수이다. fun1은 입력 값이 0보다 작거나 20보다 클 경우 예외를 발생시킨다. 예제 프로그램에서는 입력 값을 21로 하여 예외를 발생시켜 보았다. 다음 표 1은 위 프로그램의 프로파일링(profiling)한 결과이다.

함수이름	호출횟수	%시간	%메모리
nfib1	25420	99.9	99.9
nfib2	1	0.0	0.0
fun1	1	0.0	0.0
fun2	1	0.0	0.0

표 1 test1 프로그램의 프로파일 정보

fun1은 입력이 21일 때, 튜플(tuple)타입의 값 (21,*Bad*) 을 반환한다. 여기서 *Bad*는 예외 값 이라고 하자. 두 원소를 가지는 튜플 타입 중 단 하나의 원소라도 예외일 때 튜플 전체는 예외이다. 따라서 fun1이 반환하는 값은 예외 값으로 볼 수 있다. 여기까지 살펴보았을 때, fun1에서 발생한 예외는 fun2에 전달되지 않고, 즉시 fun3에 전달되어 fun3에서 처리할 수 있도록 하는 것이 가장 적합한 처리 방법이다. 하지만 프로파일링 정보에서는 fun2가 한번 호출된 것을 볼 수 있다. fun2에서는 입력 값(21,Bad)을 예외로 간주하지 않고 각각을 nfib1과 nfib2에 전달하며, nfib1은 입력으로 21을 받아 계산을 수행한다. 따라서 nfib1은 25420번 호출되어 계산을 수행한다. 결과적으로 프로그램은 불필요한 nfib1을 계산한 후 예외를 처리하게 된다. 수행시간과 메모리사용에 대한 낭비를 초래하게 된다.

위 프로그램은 예외가 다른 조각프로그램에게 전달되기 때문에 문제가 발생한다. 이 문제는 각 조각프로그램의 출력 값이 예외인지를 판단한 다음, 예외일 경우 다른 조각프로그램에게 전달하는 것을 사전에 차단하면 쉽게 해결된다.

하스켈에서 예외인지를 판단하기 위해서는 일반 타입의 값을 IO 모나드 타입의 값으로 전환해야한다. 먼저 2절에서 소개한 evaluate 함수를 이용하여 예외인지를 판단한다. 예외의 전달 유무는 프로그램을 합성하는 컴비네이터 라이브러리의 몫이다. IO 모나드의 bind 컴비네이터는 이전 계산 값이 IO 예외일 경우 예외를 전달하지 않는다. 다음 그림 8은 test1을 evaluate와 bind 컴비네이터를 이용하여 재작성한 프로그램이다.

test2는 fun1과 fun2를 IO 모나드의 bind 컴비네이터를 이용하여 합성하였다. evaluate 함수를 이용하여 fun1과fun2의 계산 값이 예외인지를 판단하였다. 하지만 예상과는 달리 프로파일링 한 결과는 표 1과 동일하였다. 다음 그림 9는 evaluate 함수의 문제점을 나타내는 프로그램이다.

```
test2 :: Int -> IO Int
test2 n
 = catch ((evaluate . fun1) n)  >>=
                  (\i -> (evaluate . fun2) i)
         (\e -> return 0)
```

그림 8 예외가 차단되는 프로그램의 예

```
tuple_ex = ((error "Ex1"), (error "Ex2"))
evaltest = catch (evaluate tuple_ex)
                 (\e -> return (0,0))
```

그림 9 evaluate에서 예외를 찾지 못하는 예

그림 9는 evaluate 함수를 이용하였지만 예외를 발견하지 못하는 예이다. tuple_ex는 두 예외 값으로 구성된 튜플이다. 예외 값을 원소로 가지는 튜플은 예외이다. 따라서 evaltest는 tuple_ex의 예외가 검출되어 (0,0) 값을 가지는 IO모나드가 되어야 하지만, 실제로는 예외 값을 가지는 IO모나드가 된다. 이와 같은 현상은 evaluate 함수가 일차원 타입(first-order type)에 대해서만 예외 값을 판단하기 때문이다.

고차타입인 튜플, 리스트 또는 사용자 데이터 타입에 대해서는 evaluate를 재 정의할 필요가 있다. 현재 라이브러리에서는 evaluate 재 정의를 위한 타입 클래스를 제공하지 않는다. 본 논문에서는 다양한 타입에서 evaluate를 이용하기 위해 Evaluate 타입 클래스를 정의하였다. 그림 10은 Evaluate 타입 클래스와 튜플, 리스트 타입에 대한 재 정의한 코드이다.

```
class Evaluate a where
 evaluate' :: a -> IO a
 evaluate' = evaluate
instance (Evaluate a, Evaluate b)
                    => Evaluate (a,b) where
 evaluate' (a,b) = do a1 <- evaluate' a
                      b1 <- evaluate' b
                      return (a1,b1)
instance Evaluate a => Evaluate ([a]) where
 evaluate' [] = evaluate' []
 evaluate' (x:xs) =  do
                     a1 <- evaluate' x
                     as <- evaluate' xs
                     return (a1:as)
test3 :: Int -> IO Int
test3 n = catch ((evaluate' . fun1) n)
                  >>= (\i -> (evaluate' . fun2) i)
          (\e -> return 0)
```

그림 10 Evaluate 클래스와 튜플, 리스트 재정의

evaluate 함수는 이미 정의되어 있기 때문에 함수 이름을 evaluate'로 하였다. 튜플과 리스트(list) 타입에 대해서는 구성 원소 중 하나라도 예외 값이 있을 때는 예외로 간주하도록 재정의 하였다. test3은 그림8의 test2에서 evaluate 함수만 evaluate'로 대체한 것이다. 표 2는 test3 프로그램을 프로파일링 한 결과이다.

함수이름	호출횟수	%시간	%메모리
nfib1	0	0.0	0.0
nfib2	0	0.0	0.0
fun1	1	0.0	2.5
fun2	0	0.0	0.0

표 2 test3 프로그램의 프로파일 정보

test3 프로그램은 evaluate' 함수와 IO모나드의 bind 컴비네이터를 이용하여 fun1에서 발생된 예외를 fun2로 전달되는 것을 사전에 차단하였다. 그 결과 fun1은 1회만 호출되며, fun2는 호출되지 않았으며, fun2에서 사용되는 nfib1과 nfib2도 호출되지 않았다.

4. 고찰

2절에서는 하스켈에서 어떻게 예외를 처리하는지 살펴보았으며, 3절에서는 프로그램 합성 관점에서 보다 효율적으로 예외처리 방법에 대해서 살펴보았다. 그 결과 지연계산 함수형언어의 특성과 예외처리의 특성은 많은 부분 상반된 관계를 가지고 있다는 것을 알 수 있었다.

지연계산의 참조투명성과 같은 우수한 특성은 예외처리 시, 시간적, 공간적인 낭비를 유발하였다. 참조투명성을 유지하기 위해 합성된 프로그램은 예외를 계속 전달하였으며, 예외 발생 후 불필요한 계산이 실제 수행되는 경우도 있었다.

3절에서 제시한 예외처리 방법은 IO 모나드를 이용하여 계산의 순서를 명시적으로 기술하는 것과 각 조각프로그램의 계산 값이 예외인지를 일일이 검사하는 것이다. 이 방법은 모든 조각프로그램을 IO 모나드로 전환해야 되는 오버헤드는 있지만, 예외가 발생할 경우 효율적으로 예외를 처리할 수 있었다. 또한 프로그래머 입장에서는 예외처리를 위해 더 많은 코드가 추가된 것처럼 보일 것이다. 하지만 모나드나 애로우와 같은 프로그램합성 인터페이스를 이용할 경우, 이 두 가지 작업은 컴비네이터 내부에 숨겨질 수 있을 것 사료된다.

최근 프로그래밍언어는 사용자에게 예외처리 인터페이스를 쉽게 제공하기 위해 try-catch와 같은 구문을 제공한다. 하스켈에서도 모나드의 편리한 사용을 위해 do 구문을 제공하고 있으며, 꽤 성공적으로 이용되고 있다. 하스켈에서 예외처리는 IO 모나드 타입을 기반으로 하고 있기 때문에 향후 예외처리를 위해 do구문을 확장하는 방안도 검토되어야 할 것이다.

5. 결론

예외처리는 견고한 프로그램을 만들기 위해 제공되는 프로그래밍 언어의 기본요소이다. 효율적인 예외처리는 예외가 발생 시 즉각 관련된 모든 프로그램은 중단하고, 대체 프로그램으로 예외적인 상황을 벗어나는 것이다.

본 논문은 지연계산 함수형 프로그램에서 효율적인 예외처리를 방법을 제시하였다. 먼저 현재 하스켈에 구현되어 있는 예외 및 예외처리 방법에 대해서 살펴보았다. IO 모나드와 evaluate'함수를 재정의 하여 예외발생 시 예외의 전달을 차단하는 방법을 소개하였으며, 실제 프로파일링을 통하여 효율성을 검증하였다.

5. 참고문헌

[1] S. P. Jones and et al, eds., *Haskell 98 Language and Libraries, the Revised 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 theories of software construction*, pp47-96, 2001.

[4] S. P. Jones, A. Reid, T. Hoare, S. Marlow, and F. Henderson, "A semantics for imprecise exceptions," in *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, (Atlanta, Georgia), pp. 25-36, May 1-4, 1999.

[5] S. van den Berg, *Extending clean with exception handling*, Master dissertation, Mathematics and Computer Science Department, Technische Universiteit Eindhoven, June 2005.

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