논•문 14-2-5

함수형 애니메이션 프로그래밍
(Functional Animation Programming)

변 석 우
경성대학교 정보과학부 컴퓨터과학과
swbyun@kyungsung.ac.kr

요 약

Haskell은 순수 함수형 언어로서 고차함수 (higher order function), 지연계산 (lazy evaluation), 정적 타입 점검 (strong type checking), 모나드 (monad) 등의 특성을 가지고 있다. 고차함수와 지연계산 기능과 더불어, 모나드, 모듈 및 클래스 타입 등의 기능이 첨가된 Haskell은 그 활용 범위를 웹 서버, 대화형 애니메이션 프로그래밍, 음악 작곡, 로봇 제어 등의 다양한 분야로 확대해 나가고 있다. 특히, 함수형 언어가 갖는 선언적 특성에 따라 프로그래머는 멀티미디어의 표현 방식 (presentation, 즉 how)에 대한 부담을 갖지 않고 멀티미디어의 내용 (contents, 즉 what)만을 고려함으로써 프로그래밍하는 추상화 (abstraction) 기법을 이용할 수 있다. 본 논문에서는 Haskell의 이러한 특성을 이용하여 그래픽스 및 애니메이션을 프로그래밍하는 방법에 대해서 논의한다.

1. 서론

고성능의 멀티미디어 하드웨어 시스템이 급속히 확산되고 있으며, 따라서 이들을 이용하는 멀티미디어 응용 프로그램들을 개발해야 하는 필요성이 대두되고 있다. 일반적으로 애니메이션을 포함한 멀티미디어 프로그래밍은 기존 프로그래밍과는 다른 다음과 같은 특성을 갖고 있다. 첫째, 멀티미디어 프로그래밍은 상부 구조의 사용자 인터페이스에서부터 하부구조에 있는 내용의 표현 (presentation)에 이르는 모든 부분들을 프로그래밍해야 한다. 하부 구조의 구체적인 부분들을 (low-level details) 프로그래밍 하는 것은 단조롭고 지루한 작업으로서 이들을 모두 일일이 처리하는 데는 많은 노력이 소요된다. 둘째, 애니메이션은 외형적으로는 연속성을 갖고 있지만, 실제 컴퓨터 내부에서 처리되는 과정은 그림의 컷들을 단계적으로 계산하는 비연속성을 가지고 있다. 비연속적인 그림의 컷들을 시간 영역에서 연속적으로 투

영하는 *실시간*적 특성을 갖는다. 셋째, 일반적으로 한 화면 내에는 여러 개체들이 표현되고 있으며 이들은 각각 독립적으로 시간에 따라 변화하며 때로는 서로 통신하고 협력하며(cooperative) 동작하는 *동시성(concurrency)*을 지니고 있다.

위의 특성 때문에 매우 단순하고 간단한 애니메이션 프로그램을 개발하는 경우라도 이것을 표현하는데 많은 비용이 들게 된다. 멀티미디어 저작 도구 (authoring tools)가 이 문제 해결에 도움을 주고 있다. 그러나, 저작 도구는 대화형 (interactive, 혹은 reactive) 형태의 애니메이션인 경우에는 이용할 수 없으므로 한계를 가질 수밖에 없으며 프로그래밍을 해야하는 필요성은 계속 존속되고 있다.

앞서 기술한 멀티미디어 프로그래밍의 특성은 기존의 프로그래밍 기법보다 훨씬 더 복잡한 프로그래밍 기법을 요구하고 있다. 본 논문에서는 순수 함수형 언어 Haskell이 갖는 고도의 추상화 기법을 적용하여 멀티미디어를 효율적으로 프로그래밍하는 기법에 대해서 논의한다. Haskell은 순수 함수형 언어로서 고차함수 (higher order function), 지연계산 (lazy evaluation), 정적 타입 점검 (strong type checking), 모나드 타입 (monad type) 등의 특성을 가지고 있다 [1]. 이러한 특성들은 그래픽이나 애니메이션 등의 멀티미디어 프로그래밍에 매우 효과적으로 기능하고 있다. 구체적으로 다음과 같은 내용에 대해서 논의한다.

첫째, Haskell은 고차함수 기능을 지니고 있다. Hughes에 의해서 제안된 대로 함수형 언어의 고차함수 기능은 프로그램 부분 (program fragments)을 합성(composition, 혹은 gluing) 하는데 중요한 역할을 한다 [2]. 애니메이션 프로그래밍에 있어서 기초 함수들을 정의하고 이 함수들을 함께 사용하는데, 이때 고차함수의 기능은 함수의 합성과 재사용을 강력하면서도 매우 간결하게 표현할 수 있다.

둘째, Haskell은 고 수준의 타입 시스템 기능을 제공한다. Hindley-Milner 형태의 타입 시스템과 더불어, 대수형 타입 (algebraic type), 모듈 (module)을 기반으로 한 추상 자료 타입 (abstract data type)과 타입 클래스 기능을 제공한다. 이러한 기능들은 프로그램의 다형성, 간결한 표현, 재사용을 가능케 한다.

셋째, Haskell은 선언적 (declarative) 특성에 따라, 애니메이션 프로그래밍에서 멀티미디어의 표현 방식(presentation, 즉 how)을 기술하지 않고 멀티미디어의 내용 (contents, 즉 what)만을 표현하는 추상화 (abstraction) 기법을 이용할 수 있으며 고급 수준의 프로그래밍 (higher-level programming)을 가능케 하고 있다.

넷째, Haskell은 선언적이지만 애니메이션 프로그래밍에서는 많은 입출력이 발생한다. 일반적으로 순수 선언적 프로그래밍에서는 입출력 및 상태 제어를 표현에 많은 어려움이 있음은 잘 알려진 사실이다. 그러나, Haskell에서는 최근 개발된 모나드 기법[3][4]을 이용함으로써 입출력을 비교적 쉽고 자연스럽게 표현할 수 있다.

본 논문에서는 이와 같은 고수준의 Haskell 프로그래밍을 그래픽스와 애니메이션에 적용하는 기법에 대해서 설명한다. Haskell의 풍부한 타입 시스템을 이용하여 비트맵으로부터 도형의 색, 도형의 위치 이동, 도형의 축소/확대, 도형의 합성 (union), 중첩 (intersect), 도형의 안/밖 구분 (complement), 시간 변화에 따른 화상 변화 등의 애니메이션에 필요한 자료구조를 정확

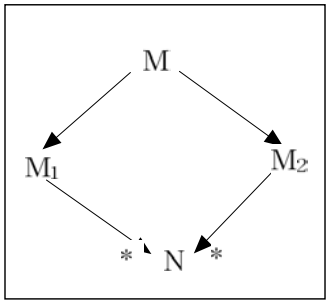
하게 표현할 수 있으며, 이를 바탕으로 마우스 클릭에 의해 제어되는 그래픽 프로그램과 동영상 프로그램을 논리적으로 간결하게 표현할 수 있다. 함수형 프로그래밍에서 고차 함수 기능을 이용하여 이미 정의된 함수(즉, 프로그램 부분)들을 입력 인수로 받아들여 새로운 기능을 정의하는 함수들을 컴비네이터라고 부르는데, 대표적인 컴비네이터로서 map과 fold가 있다. 이 두 컴비네이터와 함께, 애니메이션을 위한 새로운 컴비네이터 over를 소개한다.

2. Haskell의 특성

2.1. Rewriting

(1) CR (Church-Rosser) 특성

CR은 rewriting 이론의 가장 근본적인 특성이다. 일반적으로 이 특성은 [그림 1]로서 표현되고 있다.



(그림1) Church-Rosser 특성

위 그림에서 두 개의 레덱스 (redex)를 포함하고 있는 텀 M은 선택된 레덱스의 rewriting 결과 각각 M_1과 M_2의 상태로 변환할 수 있는데, 이 경우 M_1과 M_2 각각에 대하여 rewrite하여 도달할 수 있는 텀 N이 반드시 존재한다. 이 원리에 따라 N이 정규형 (normal form)인 경우 그것은 계산 순서에 상관없이 유일한 형태로 존재한다. 레덱스의 선택이 결과 값에 영향을 주지 않게 되므로 여러 레덱스들을 동시에 처리할 수 있는 병렬성이 내재되어 있다.

(2) 정규형 전략 (normalizing strategy)

예를 들어, f A x = 1, B = B와 같은 두 룰이 정의되었을 때, "f A B" 의 식을 계산한다고 가정하자. "f A B"에서는 두 개의 레덱스 "f A B"와 "B"가 존재한다. CR 특성에 따라 이 중에 어느 것을 먼저 선택하더라도 값이 달라지지는 않지만, "B"를 계속해서 선택하면

f A B = f A B = ...

로 되어 결과 값에 도달할 수 없다. 그러나, "f A B"를 선택하면

f A B = 1

이 되어 결과 값에 도달할 수 있다. 위의 경우, "B"를 선택하는 것은 정규형을 계산하는데 아무런 도움을 줄 수 없다. 정규형에 도달할 수 있는 레덱스 선택 기법을 정규형 전략이라 하는데, Haskell과 같은 함수형 언어에서는 정규형 전략을 갖고 있다. 정규형 전략이 존재한다는 것은 계산 과정을 자동적으로 추론할 수 있음을 의미하므로, 이 전략을 프로그래밍 시스템에 구현함으로써 프로그래머가 계산 과정을 기술하지 않더라도 시스템이 이것을 자동적으로 처리해 준다. 흔히, 함수형 프로그래밍 등의 선언적 언어의 프로그래밍은 "what만을 기술하고 how는 기술하지 않아도 된다" 라는 것은 이러한 정규형 전략에 따른 것이다.

2.2. 고차 함수 기능

Haskell에서 함수는 first-class citizen으

로서 함수의 인수나 복귀(return) 값으로서도 이용될 수 있다. 예를 들어, 함수 map은 다음과 같이 정의될 수 있다.

```
map f [ ] = [ ]
map f (x:xs) = (f x) : (map f xs)
```

map은 함수와 리스트를 받아 들여 리스트의 각 구성원에게 함수를 적용하여 변형시키는 일을 수행한다. 예를 들어 제곱근을 구하는 함수 sqrt와 리스트 [4, 9, 16]이 주어졌을 때, map sqrt [4, 9, 16] = [2, 3, 4]가 된다. sqrt는 unary 함수이지만 고차 함수의 원리를 적용하여 sqrt에 대한 인수를 표현하지 않은 채 그냥 sqrt로서만 기술할 수 있는 특징이 있다. .

함수형 언어에서 두 함수 f와 g를 합성하는 것은 (g . f)로서 표현된다. 이 합성된 함수를 d에 응용한 (g . f) d 에서는 f가 d를 처리한 결과 값이 g에 입력된다. 지연 계산의 기능에 따라 f가 계속 d를 계산하는 것이 아니라 g가 계산하기를 원할 때 비로소 f가 d를 계산하도록 한다. 즉, 지연 계산은 일련의 계산 과정에 대한 정확한 동기화(synchronization)를 보장하며, 반드시 필요한 경우만 계산이 진행되도록 한다. 따라서 d가 무한 형태의 값이 되더라도 적절하게 계산할 수 있다 [2].

2.3. 타입 시스템

최근 개발된 대부분의 함수형 언어는 Int, Float, Bool, List 등의 기본 타입과 함께 Hindley-Milner 형태의 타입 시스템을 보유하고 있다. Hindley-Miler 형태의 타입에서는 타입의 정의에 변수가 이용될 수 있다.

```
id : a -> a
id x = x
```

와 같은 id 함수 타입의 정의에 변수 a가 이용되었다. 이 경우 id 함수는 입력으로 모든 타입의 식을 받아들일 수 있다. 즉, id 1 = 1, id [1, 2, 3] = [1, 2, 3], id id = id 등과 같이 id는 입력으로서 정수, 리스트, 함수 등의 다양한 종류를 입력받을 수 있다. 이와 같이 타입 변수를 이용하여 정의되는 함수를 polymorphic 함수라 부른다.

Haskell에서는 polymorphic과 overload 두 종류의 다형성을 이용하고 있다. overloading 기법의 다형성은 타입 클래스에 의해서 표현된다. 예를 들어, 주어진 두 식이 같은 값을 갖는가에 대한 여부를 판단하는 Eq 클래스의 (==) 오퍼레이터는 다음과 같이 정의된다.

```
class Eq a where
(==) :: a -> a -> Bool
```

(==)는 임의의 같은 타입을 갖는 두 식을 받아서 True나 False의 결과 값을 준다. 이 두 식은 정수, 리스트, 실수 등 다양한 형태를 지닐 수 있는데, 각 경우마다 알고리즘이 다르며 따라서 (==)를 구현하는 형태가 달라지게 된다. 각 구현 형태는 instance로서 표현되는데, 예를 들어 a가 Int인 경우

```
instance Eq Int where
(==) :: Int -> Int -> Bool
(==) x y = primEqInt x y
```

의 형식을 갖는다. 여기서 오른쪽에 있는 primEqInt는 미리 정의된 함수이다. 모든 타입에 대해서 (==)에 대한 구현 방식을 정의할 수 있으면 그에 대한 (==)를 overload 형태로 정의할 수 있다. 기타 algebaic type 및 구체적인 내용들은 [5], [6]을 참조하기 바란다.

2.4. 흐름제어와 모나드

함수형 언어는 CR 특성에 따라 계산 순

서를 자유롭게 결정할 수 있지만, 반대로 계산을 순차적으로 처리할 경우에는 이것을 표현하기가 어려운 문제점을 가지고 있다. 순차적 계산을 요구하는 대표적인 예로서 입출력과 효과 (effect)의 특성을 갖는 경우가 있다. 예를 들어, (display 'a', display 'b')과 같이 화면에 연이어 글자를 출력하는 두 레덱스가 주어진 경우를 고려하자. 두 레덱스 중에서 전자를 먼저 계산하면 "ab"로 출력될 것이며 후자를 먼저 출력하면 "ba"가 된다. 입출력 프로그래밍에서는 일반적으로 출력 결과가 "ab"가 되어도 좋고 "ba"가 되어 좋은 것이 아니다. 따라서 이 경우에서는 CR의 원리가 성립되지 않으며 계산을 어떤 특정 순서에 따라 순차화 (sequentialize) 시키는 것이 필요하다. 그러나 잘못 처리되는 순차화 기법은 CR을 성립시키지 못할 수 있으므로 이 둘의 특성을 동시에 지원하는 데는 여러 어려움이 따르고 있다. 최근 모나드가 이 문제에 대한 해결 방법을 제시하고 있다.

모나드에서는 전통적인 '값들의 집합' 개념의 타입과 더불어 '액션 (혹은, 계산)을 의미하는 타입'이라는 개념을 새로 도입하였다. 타입 a에 대하여 타입 구성자 m을 이용한 새로운 타입 (m a)은 액션 m을 수행한 후 a 타입의 결과 값을 얻게 된다. 예를 들어, m이 입출력을 의미하는 IO일 경우 (IO Int)는 입출력을 수행한 결과 값으로서 정수 값을 얻게 됨을 의미한다. 입출력을 통하여 하나의 문자를 입력받는 함수 getChar는 (IO Int)의 타입을 가지며, 반대로 한 문자를 출력하는 함수 putChar는 (IO ()) 타입을 갖는다. 여기서 ()는 unit 타입으로서 무의미한 값을 뜻한다.

Haskell에서 모나드는 클래스 타입 Monad로서 정의되어 있다.

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

어떤 한 프로그램의 계산을 모나드 형태로 제어를 하기를 원한다면 모나드 클래스에서 타입 m에 대한 >>=와 return을 정의하는 것이 필요하다. 예를 들어, 입출력을 모나드 방식으로 프로그래밍하기 위해서는 위의 클래스 타입에 있어서 모나드 m에 대한 인스턴스에 대한 다음과 같은 타입을 갖는 함수 >>=와 return이 정의되어야 한다.

```
instance Monad IO where
(>>=)  :: IO a -> (a -> IO b) -> IO b
return  :: a -> IO a
```

모나드를 구성하는 두 개의 기본 함수는 return과 >>= 이다. return은 (unit라고도 불림) 어떤 인수를 받아서 그것을 결과 값으로 전달하는 *단순한 계산*을 하고 있으며 >>= (bind, 바인드라고 불림) 오퍼레이터는 두 개의 계산을 받아들여 첫 번째 계산의 결과를 두 번째 계산의 인수로 전달시킴으로써 첫 번째 계산과 두 번째 계산을 *순차적*으로 합성시키는 역할을 하고 있다. (g>>=h)는 g의 액션이 h보다 선행됨을 *명시적*으로 표현하고 있다. 예를 들어, IO 모나드인 경우

```
getChar :: IO Char
putChar :: Char -> IO ()
```

의 함수가 주어졌을 때, (getChar >>= putChar)에서 getChar는 IO를 수행한 결과 값으로 문자 타입의 값을 얻게 되며, >>= 오퍼레이터는 이 값을 putChar의 입력 값으로 전달하는 역할을 한다. putChar는 주어진 문자를 입출력을 통하여 표준 출력 (standard-out)하게 된다. 첫 번째 계산을 수

행한 결과를 두 번째 계산의 입력으로 사용하지 않는 경우를 고려하여 >> 오퍼레이터가 다음과 같이 정의되다.

```
g >> f  =  g >>= \_ -> h
```

이와 같이, 바인드는 두 개의 함수를 입력으로 받아서 이 두 함수의 계산이 순차적으로 진행되도록 하는 컴비네이터이다. 계산의 순차화를 명령형 프로그래밍과 유사한 형태로 표현할 수 있도록 해 주는 것으로 do 구문이 있다. 다음과 같이 do는 단지 바인드의 표현을 바꾼 것에 불과하다.

```
do { x <- e; s}  =  e >>= \x -> do {s}
do { e; s}       =  e >> do {s}
do { e }         =  e
```

예를 들어, (getChar >>= putChar)를 do 구문을 이용하여 표현하면 다음과 같다.

```
do {x <- getChar;
    putChar x}
```

모나드는 계산 순차화 외에 하부 구조의 구체적인 부분들을 (low-level details) 감춤으로써 고수준의 추상화 (higher-level abstraction)를 가능케 한다. 예를 들어, 사칙연산에서 주어지는 값을 연산하기 전에 주어지는 값이 잘못된 값인 지의 여부를 판단하고 올바른 값에 대해서만 계산을 하도록 사칙 연산의 기능을 확대하는 경우를 고려하자. Haskell에서는 이런 목적을 위해 다음과 같은 Maybe 타입을 이용하고 있다.

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

Maybe 타입은 Just와 Nothing의 두 타입 구성자를 이용하고 있다. 계산 과정이 문제없이 진행된 경우 그 값은 Just로서 표현되며, 만약 올바른 계산을 하지 못하였을 경우 (예를 들어, 어떤 값을 0으로 나누었을 때) 그 값은 Nothing으로서 표현된다. 이 결과 값을 이용하는 측에서는 결과 값이 Just의 형태로 되어있는지 혹은 Nothing의 형태로 되어 있는지를 보고 계산이 제대로 이루어졌는지의 여부를 판단할 수 있다.

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

이 add 함수에서는 주어진 두 인수가 모두 Just인 경우는 더하기를 수행하지만 만약 이 중에 하나라도 Just의 형태가 아닌 경우라면 결과 값으로 Nothing을 전달한다. 이 상황을 Maybe 모나드를 이용하여 표현할 수 있다. 먼저 Maybe 모나드를 다음과 같이 표현한다.

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

이렇게 정의된 Maybe 모나드를 이용하여 add 함수는 다음과 같이 새롭게 정의될 수 있다.

```
addM :: Maybe Int -> Maybe Int
        -> Maybe Int
addM x y = do a <- x
              b <- y
              return (a+b)
```

addM 함수는 Maybe 타입의 인수들을 받아서 계산하지만 주어지는 값의 올바름 여부를 체크하는 하부구조의 구체적인 부분들을 표현함이 없이 (이 과정은 Maybe 모나드에 기술되어 있음), 단순히 더하는 과정만을 표현함으로써 프로그램을 간결하고 명확하게 표현하고 있다. add와 addM을 비교할 때,

add에서는 두 인수의 올바름 (즉, Just) 여부가 명시적으로 표현되고 있는 반면에 addM에서는 이들이 표현되지 않고 있다. 데이터를 계산하는 순서 측면에서 볼 때, add에서는 두 인수를 처리하는 순서에 아무런 제약이 없지만 (즉, 이 두 인수의 처리를 병렬로 수행할 수 있다), addM에서는 반드시 첫 번째 인수에 대한 검토가 수행된 후 두 번째 인수에 대한 검토가 수행된다. 이와 같이, 모나드는 계산을 순차화시키며 프로그래밍의 추상성을 높이는 중요한 역할을 한다.

3. 그래픽 애니메이션

3.1. 도형들의 데이터 타입

이 절에서는 사각형, 삼각형, 그리고 다각형으로 구성된 도형들에 대한 그래픽 처리 기법을 소개한다. 본 절에 소개되는 내용은 Hudak 프로그램 코드 [6] 중 일부로서 핵심적인 내용만을 소개하기로 한다.

먼저 도형들의 모습은 다음과 같이 algebraic 타입을 이용하여 정확하고 간결하게 표현될 수 있다. 도형들의 모습은 Shape, Region, Picture의 세 가지 측면에서 표현되고 있다. Shape은 기초적인 형태의 모습을 실수 값으로 표현하고 있다. 사각형, 타원, 직삼각형의 크기는 모두 두 개의 실수 값으로 정의될 수 있으며, 일반적인 다각형은 변의 크기로 기술할 수 없으므로 이들은 윈도우 상의 픽셀 좌표로 표현된다. 다음 Shape의 정의에서 Side, Radius는 실수 값을 의미하며, Vertex는 정수 값으로서 픽셀 좌표를 나타내고 있다. 각 도형을 나나태기 위한 타입 구성자 Rectangle, Ellipse, ReTriangle, Polygon이 사용되고 있다.

```
data Shape =   Rectangle Side Side
             | Ellipse Radius Radius
             | RtTriangle Side Side
             | Polygon [Vertex]
```

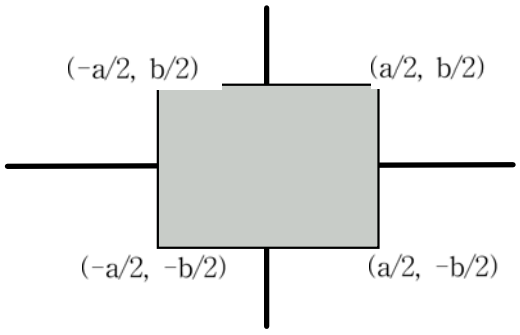
Region은 Shape을 확장시킨 것이다. Region에서는 위에 정의된 Shape의 정보를 나타내기 위한 Shape, 도형의 위치를 표현하는 Translate, 도형의 크기를 표현하는 Scale 등의 타입 구성자들이 이용되고 있다.

```
data Region = Shape Shape
                  -- primitive shape
  | Translate Vector Region
                  -- translated region
  | Scale Vector Region
                  -- scaled region
  | Complement Region
                  -- inverse of region
  | Region `Union` Region
                  -- union of regions
  | Region `Intersect` Region
                 -- intersection of regions
  | Empty         -- empty region
```

Region의 첫 번째 정의되고 있는 (Shape Shape)에서 왼쪽의 Shape은 타입 구성자를 의미하며 오른쪽 Shape은 위에서 정의된 타입 Shape을 의미한다. 예를 들어, 사각형 모습의 Shape을 Region 타입으로 표현할 경우에 모습은 (Shape (Rectangle 3.0 4.0))과 같은 형태로 표현된다. Translate 에서는 Vector 정보가 이용되는데, Vector는 (x, y) 값을 실수 형태로 표현하는 쌍으로서 도형의 모습을 각각 x와 y 만큼의 비율로 확대함을 의미한다. Haskell에서 함수나 타입 구성자를 prefix 대신 infix 형태로 표현할 때는 ` ` 로 표현한다. 위의 Region 정의에서 Region `Unition` Region은 prefix 형태의 (Union Region Region)과 같다. Region의 타입 구성

자 중에서 Complement, Union, Intersect는 각각 어떤 한 도형의 바깥 부분, 두 도형이 합쳐진 모습, 두 도형의 포개진 부분의 모습을 표현하는 역할을 한다. 이 타입 구성자들은 윈도의 한 좌표가 어떤 도형 내에 속해 있는지의 여부를 판가름하는 Predicate를 정의할 수 있도록 해 준다. 이 Predicate를 정의함에 따라서 도형을 매우 간단하고 편리하게 처리할 수 있다.

한 Region 형태의 도형이 주어졌을 때 어떤 한 좌표가 그 도형 안에 포함되어 있는지의 여부를 결정하는 Predicate 함수 containsR :: Region -> Coordinate -> Bool을 정의할 수 있다. 이것을 정의하기 위해서는 먼저 Region 보다 간단한 형태인 Shape에서 이것을 판가름하는 containsS :: Shape -> Coordinate -> Bool을 정의하는 것이 필요하다. Shape은 Rectangle, Ellipse, RtTriangle, Polygon으로 구성되어 있으므로 각 도형의 특성에 따라 주어진 좌표 (x, y)가 그 속에 포함되어 있는지를 판가름할 수 있다. 예를 들어, 다음 [그림 2]처럼 사각형의 영역을 표현할 수 있다.



(그림 2) 사각형의 영역

[그림 2]에서 세로의 길이가 a, 가로의 길이가 b인 사각형인 경우 이들의 모서리에 해당하는 좌표는 다음 그림과 같이 (a/2, b/2), (-a/2, b/2),(a/2, -b/2),(-a/2, -b/2)이며, 이때 Rectangle인 경우의 constainsS는 다음과 같이 정의된다.

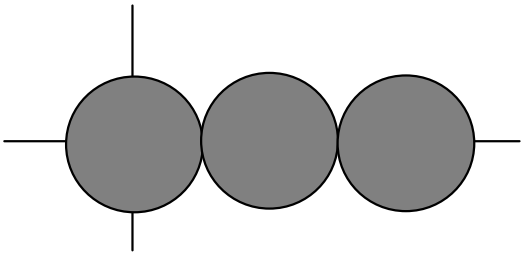
```
containsS (Rectangle a b) (x, y)
  = (-a/2 <= x) && (x <= a/2)
    && (-b/2 <= y) && (y <= b/2)
```

이와 유사한 방법으로 삼각형, 타원, 다각형에 대해서도 constainsS를 정의할 수 있다. Region의 Translate는 위치를 이동시키는 역할을 한다. Haskell이 갖는 list comprehension 기능을 이용하면 아래 [그림 3]의 모습의 도형을 다음과 같이 매우 간략하게 표현할 수 있다.

```
oneCircle = Shape (Ellipse 1 1)
manyCircles = [Translate (x, 0) oneCircle |
                  x <- [0, 2 ..]]
threeCircles = fold Union Empty
                    (take 3 manyCircles)
```

원은 장축과 단축의 길이가 같은 타원의 특수한 경우이다. 따라서 oneCircle은 반지름이 1인 원을 의미한다. manyCircles는 List comprehension으로서 정의되어 있는데, 여기서 x는 0, 2, 4, 6, ... 형태의 무한 값을 갖는다. 따라서 manyCircles는 oneCircles를 좌표 (0, 0), (2, 0), (4, 0) ... 상에 무한개 위치시킨 원들을 의미한다. threeCircles에서 (take 3 manyCircles)는 manyCircles 중 처음 3개를 선택한다. 따라서 선택된 원들은 (0, 0), (2, 0), (4, 0) 위치에서 그려진 3개의 원들의 리스트이다. fold 함수와 함께 사용되고 있는 Union 함수는 이들을 Union 시킨 데이터 값들의 형태로 만든다. fold는 Union을 리스트 원소들 각각에 재귀적으로 적용시킨다. Empty는 도형의 아무런 모습도 표현하지 않

는 것으로서 fold의 init 인수로서 사용되고 있다. 따라서 위의 threeCircles의 모습은 다음 그림과 같은 형태로 표현된다.



(그림 3) 3개의 원 그리기

containsS 함수를 사용하면 containsR은 쉽게 표현될 수 있다. Region이 Shape인 경우 containsR은 containsS와 거의 같으며, Translate, Scale, Complement, Union, Intersect의 각 경우에 대해서 좌표 정보를 이용하여 포함여부를 계산할 수 있다. 자세한 사항은 [6]을 참조하기 바란다.

앞서 정의된 Region의 내용을 확장하여 Picture를 정의한다. Picture는 아래와 같이 Region에 색상에 대한 정보가 덧붙여 진 것(Region Color), 그리고 여러 개의 Region들이 (부분적으로) 포개져 표현되는 현상을 표현하고 있다.

```
data Picture = Region Color Region
             | Picture `Over` Picture
             | EmptyPic
```

그래픽의 내용을 화면으로 출력할 때는 시스템에서 제공하는 라이브러리를 사용하게 된다. 라이브러리 각각의 형태에 따라 달라질 수 있지만, 출력은 대개 다음과 같은 형태로 진행된다.

1. 먼저 화면에 새로운 윈도우를 만든다 (open). 이때, 그 크기와 출력될 타이틀이 주어진다.

2. 주어진 윈도우에 그래픽 내용을 출력한다.

3, 윈도우의 사용을 종료할 때는 윈도우를 닫는다 (close).

그래픽의 출력시 그림들이 중첩될 수 있는데, 이 내용은 타입 구성자 Over로서 표현된다. 중첩된 그림의 표현은 그림의 출력 순서 제어된다. 위의 Picture의 내용을 출력하는 함수 drawPic은 다음과 같이 정의된다.

```
drawPic  :: Window -> Picture -> IO ()
drawPic w (Region c r) =
        drawRegionWindow w c r
drawPic w (p1 `Over` p2) =
         do drawPic w p2; drawPic w p1
drawPic w EmptyPic = return()
```

Picture가 Color를 갖는 Region인 경우는 그 내용대로 화면에 출력하며, EmptyPic인 경우는 아무런 내용도 출력하지 않는다. Picture가 (p1 `Over` p2) 된 형태라면 p2를 먼저 출력한 후 p1을 출력한다. 따라서, 만약 p1과 p2의 일부분이 겹쳐졌을 때는 p2의 겹쳐진 부분이 보이지 않게 된다. 위의 Over에 대한 drawPic 정의에서 출력 순서가 do 구문에 의해서 순차화적으로 표현되고 있다.

3.2. 주요 컴비네이터들

함수형 프로그래밍에서의 컴비네이터들은 함수들을 합성시키는 기능을 한다. 대표적 컴비네이터들인 map, fold, do, sequence_ 등이 그래픽 프로그래밍에 적용되는 사례를 소개한다. (do에 대해서는 이미 충분한 설명이 이루어졌으므로 생략하기로 한다.)

(1) map

map은 가장 널리 사용되는 컴비네이터들 중에 하나이다. 앞서 map의 기능은 충분히

설명하였다.

```
conCircles  = map circle [2.4, 2.1..0.3]
circle r = Ellipse r r
coloredCircles = zip [Black, Blue, Green,
Cyan, Red, Magenta, Yellow, White]
                    conCircles
```

circle은 원을 타원 형태로 표현한 것이다. [2.4, 2.1..0.3]로 표현된 리스트는 -0.3 (= 2.1 - 2.4) 만큼씩 감소하여 만들어지는 8개 원소 [2.4, 2.1, 1.8, 1.5, ..., 0,3], 즉 반지름이 2.4에서부터 0.3인 8개의 원을 의미한다. zip은 zip :: [a] -> [b] -> [(a, b)]로서, coloredCirles는 [(Black, 2.4), (Blue, 2.1), ..., (White, 0.3)]이 된다. 즉, 각 원의 반지름과 색을 나타내고 있다.

(2) seqeunce_

앞서 do를 이용하여 입출력을 프로그래밍하는 예를 소개하였다. do 구문은 프로그램의 위에서부터 아래쪽으로 프로그램을 순서적으로 수행한다. 입출력에 있어서 이와 유사한 기능을 하는 sequence_ 함수가 있다. sequence_는 입출력을 수행하는 액션들의 리스트를 입력으로 받아서 그들을 순서대로 하나씩 연속하여 처리하는 기능을 한다. 다음 예에서 sequence_ 는 actionList에 있는 출력문들을 수행한다.

```
sequence_ :: [IO a ] -> IO ( )
actionList = [putStr "Hello World\n",
              writeFile "test" "Hello",
              putStr "file written."]
sequence_ actionList
```

여기서 주목할 점은 일반 데이터뿐만 아니라 액션들 또한 리스트의 원소가 되고 있다는 점이다. 이론적으로 정수나 문자와 같은 데이터는 상수 (constants, 즉 0-ary 함수)로서 고차함수 원리가 적용되는 함수형 언어에서는 상수와 함수를 다르게 분리하여 처리하지 않는다. 위의 예에서 actionList는 프로그래머에 의해서 정의되었지만, 다음과 같이 액션을 프로그램으로 생성해 낼 수도 있다.

```
drawShapes :: Window -> [(Color, Shape)]
                  -> IO ( )
drawShapes w css
 = sequence_ (map aux css)
aux (c, s) = draw w (withColor c
                (ShapeToGraphic s))
```

drawShapes는 윈도와 색과 Shape의 튜풀 형태로 주어진 인수를 받아서 이들을 화면에 출력하는 기능을 한다. 위에 정의된 원들 coloredCircles은 drawShapes에 의해서 화면에 출력된다. drawShapes에서 한 개의 Shape을 화면에 출력시키는 일은 draw 함수 (이미 정의됨)가 담당하며, (map aux css)는 색과 Shape으로 구성된 리스트 원소들을 출력시키기 위한 액션들을 리스트로 만들고, sequence_는 이 액션 리스트를 화면에 출력된다. 이와 같이 (입출력 등의) 액션들을 프로그램에 의해서 생성함으로써 매우 간결한 프로그래밍을 가능케 한다.

(3) fold

여러 원소들을 포함하는 리스트에 어떤 함수가 반복하여 적용되는 기능을 하는 함수로서 앞서 논의된 map과 더불어 fold가 있다. map과 fold는 몇 가지 측면에서 차이점을 보이고 있는데, 우선 map은 입력으로 리스트를 받고 결과로서 리스트를 출력하지만, fold는 리스트를 입력으로 받아서 리스트가 아닌 형태로 출력한다. 이런 특성 때문에 fold를 collapse 함수라고 부르기도 한다.

```
fold :: (a -> b -> b) -> b -> [a] -> b
fold op init [ ] = init
```

```
fold op init (x:xs) = op x (fold op init xs)

fold op init (x1 : x2 : ... : xn : [ ])
==> op x1 (op x2 ( .... (op xn init) ...))
```

위의 정의된 대로 fold는 이진 함수, init, 그리고 list의 3개의 인수를 받는다. 여기서 init은 각 함수에 따라서 달라진다. 예들 들어, 사칙 연산의 (x + init = x)가 되는 init은 0이며, (x * init = x)가 되는 init은 1이다. 즉, 더하기와 곱하기 함수에 대한 init은 각각 0과 1이다. 위의 여러 도형들을 합집합의 모습으로 출력하기 위한 manyCircles 함수에서 사용되는 Union에서 (x `Union` init = x)가 되는 init은 Empty이다.

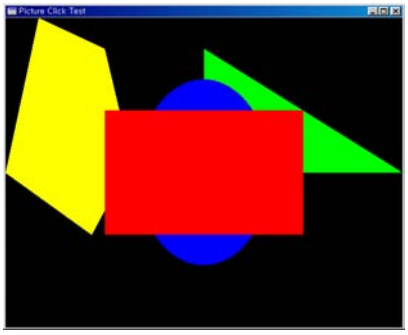
```
fold (+) 0 [1, 2, 3]
  = 1 + (2 + (3 + 0)) = 6

fold (*) 1 [1, 2, 3]
  = 1 * (2 * (3 * 1))  = 6
```

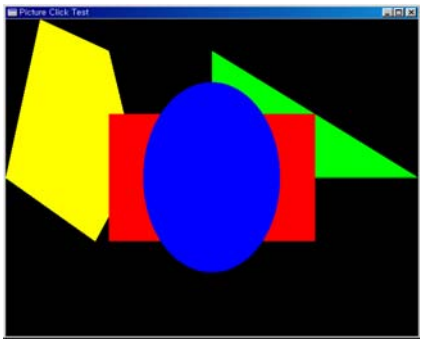
3.3. 도형 그리기

다음 [그림 4]는 중첩되어 디스플레이 된 여러 도형들 중 어느 하나를 마우스로 클릭하였을 때 클릭된 지점에 있는 도형을 맨 위로 옮기는 모습을 보여준다. 클릭된 지점을 좌표로서 파악하는 것은 Haskell 라이브러리 함수이다. 마우스에 의해서 지정된 좌표를 읽어 들이면 이 지점이 어떤 도형의 내부에 포함되어 있는가를 판정하는 것은 이미 앞서 언급한 containsR 함수에 의해서 처리될 수 있다. containsR를 이용하여 맨 위에서부터 아래쪽 방향으로 그 지점을 포함하는 도형을 찾아 이들 맨 위에 디스플레이 함으로써 원하는 프로그램을 완성시킬 수 있다.

[그림 4]는 약 10줄 정도의 Haskell 코드로서 간단히 표현될 수 있다. 이처럼 간결한 코드를 가능케 하는 것은 앞서 언급한 고차 함수 기능, Hindley-Milner 타입 및 다형성, 대수형 타입, 컴비네이터 등의 특성이 상호 대화형 프로그램일 경우에도 완벽하게 기능하기 때문이다.



< 포개져서 디스플레이된 도형들>



<클릭된 타원이 맨 위로 이동된 모습>

(그림 4) 도형 그리기 및 변환시키기

4. 애니메이션 프로그래밍

애니메이션 프로그램은 지금까지의 정적인 화면과는 달리 시간에 따라 화면이 달라진다. 영화와 마찬가지로 동영상을 만드는 방법은 인간이 느끼는 '시각의 잔상' 원리를

이용한다. 즉, 1초에 20내지 30개 정도의 프레임 (frame)을 영사함으로써 인간은 '연속으로 보여지는 정지된 화면'을 마치 움직이는 화면처럼 느끼게 된다. 정적인 그래픽 화면과는 달리, 움직임의 효과를 내기 위해서는 한 프레임을 디스플레이 할 때 그 전 상태 프레임의 일부를 지워야 한다.

4.1 애니메이션 타입

애니메이션의 가장 기본이 되는 타입 Animation은 모든 객체 a에 대해서 실수 값으로 된 Time에 따라 변화는 객체 a로 정의된다.

```
type Animation a = Time -> a
type Time = Float
```

여기서 a는 타입 변수로서 애니메이션 내용의 데이터 타입으로 실례화 된다.

```
revolvingBall :: Animation Region
revolvingBall t =
    let ball = Shape (Ellipse 0.2 0.2)
    in Translate (sin t, cos t) ball
```

이 함수는 장축과 단축의 반지름이 모두 0.2 inch 인 타원 (즉, 0.2 inch 반지름의 원)을 좌표 (sin t, cos t)에 위치시킨다. 시간 t가 변함에 따라서 (sin t, cos t)의 값은 (0, 1)에서 시작하여 x 값은 점차 증가하고 y 값은 그에 비례하여 감소하게 되어 (1, 0)의 형태로 된다. 즉 (0, 1), (1, 0), (0, -1), (-1, 0), (0, 1)을 반복하여 통과하는 원운동을 하게 된다.

4.2. 애니메이터

위에 소개된 두 애니메이션 함수는 애니메이션을 위한 데이터 값을 만들지만 이들만으로 애니메이션이 이루어지지는 않는다. 이 내용을 시간에 맞추어 화면에 출력시키기 위해서 animate라는 함수가 정의된다.

```
animate :: String -> Animation Graphic
             -> IO ( )
```

animate는 제목 (String)과 Animation Graphic 타입의 데이터를 화면으로 출력한다. 앞서 소개된 Shapes, Region, Picture와 더불어 Graphic 타입이 위에 만들어진 두 데이터는 Animation Shape과 Animation Region 타입을 갖고 있으므로 이를 각각 Animation Graphic 형태로 변형시켜야 한다. Graphic 화면 출력에 요구되는 상세한 정보들을 담고 있다.

picToGraphic :: Picture -> Graphic 는 Picture 형태로 구성된 데이터를 Graphic으로 변환시키는 함수이다. Picture 형태의 planets :: Animation Picture 이라는 애니메이션은 animate 함수와 picToGraphic을 이용하여 다음과 같이 화면에 그릴 수 있다.

```
animate "Animated Picture" (picToGraphic
. planets)
```

```
animate title anim
 = runGraphics $$
   do w <- openWindowEx title
           (Just (0,0)) (Just (xWin,yWin))
      drawBufferedGraphic (Just 30)
      t0 <- timeGetTime
      let loop =
       do t <- timeGetTime
        let ft = intToFloat
              (word32ToInt (t-t0)) / 1000
          setGraphic w (anim ft)
          getWindowTick w
          loop
      loop
```

(1) openWindowEx - 주어지는 데이터에 따

라 새로운 윈도를 생성한다.
(2) drawBufferedGraphic - 동영상 디스플레이는 그전 화면의 내용을 지운 후 새로 디스플레이 해야 한다. 여기에는 exclusive-or 함수를 사용하는 방법과 버퍼를 사용하는 방법을 고려할 수 있는데, 위 프로그램에서는 디스플레이 할 내용을 버퍼에 저장하고 이 내용을 주기적으로 화면에 출력하는 방식을 이용하고 있다. 이 함수는 30 millisecond 주기로 디스플레이하도록 지정하고 있다.
(3) getWindowTick :: Window -> IO ( ) - 앞서 display를 30 millisecond 마다 실행하기로 하였으므로, 논리적인 시계의 기본 단위인 "tick"은 30으로 정의된 셈이다. 이 함수는 화면에 디스플레이 될 내용을 매 tick마다 출력시키는 역할을 한다.
(4) timeGetTime :: IO Word32 - 하드웨어 시스템에 있는 시계에 의해서 주어지는 시간을 읽어 온다.
(5) setGraphic :: Window -> Graphic -> IO ( ) - 화면에 그려질 그래픽 이미지를 set 시키는 역할을 한다. 이 함수는 원래 화면에 있던 모든 데이터를 없애고 새로운 이미지 화면을 구축한다.

위의 animate는 loop로 정의된 내용을 반복 수행하도록 정의되어 있다.

4.3 Behavior 타입과 다형성 함수

애니메이션을 구성하기 위해서는 각 시간별 프레임을 계산해야 하지만, 시간을 기준으로 각 프레임을 계산하는 것은 매우 번거로운 일이다. 시간에 따라 자동적으로 애니메이션 될 여러 내용들을 합성할 수만 있다면 시간 위주가 아닌 애니메이션 될 내용 부분들 위주로 애니메이션을 정의할 수 있으므로 애니메이션 프로그래밍이 매우 편리해 진다. 이때 이들을 함께 표현하는 타입 구성자 Over를 이용하는 것이 편리하다. 또한, 여러 형태의 데이터 타입을 한 화면에 합성시키는 다형성 함수 over가 유용하다. 여기서는 애니메이션에 다형성 함수를 적용하는 기법에 대해서 소개한다. 먼저 시간에 따라 변화하는 값을 표현하는 애니메이션에 대한 타입을 선언한다.

```
newtype Behavior a = Beh (Time -> a)
```

앞서 정의된 animate를 이용하기 위한 인터페이스 함수 animateB가 다음과 같이 정의된다.

```
animateB :: String -> Behavior Picture
              -> IO ()
animateB s (Beh pf)
      = animate s (picToGraphic . pf)
```

한 도메인 a를 도메인 (T a)으로 변환시키는 것을 "lifting"이라 한다. 앞서 여러 함수들을 Behavior로 lift하는 것이 필요하다. Lifting은 함수의 인수의 수 (arity)에 따라 다음과 같이 정의할 수 있다.

```
lift0 :: a -> Behavior a
lift0 x = Beh (\t -> x)

lift1 :: (a -> b) ->
         (Behavior a -> Behavior b)
lift1 f (Beh a)  = Beh (\t -> f (a t))

lift2 :: (a -> b -> c) -> (Behavior a
           -> Behavior b -> Behavior c)
lift2 g (Beh a) (Beh b)  = Beh (\t -> g
(a t) (b t))
```

유사한 방법으로 3개나 그 이상의 인수를 갖는 함수들을 lifting하는 함수들을 정의할 수 있다. +의 인수의 수는 2이고, 조건문은 3개의 인수의 수 (즉, Bool, then일 경우 수행되는 함수, else일 경우 수행되는 함수)를 가

지고 있으므로, 이들은 각각 lift2와 lift3에 의해서 lift될 수 있다. 이 lift 함수들을 이용하여 Num, Fractional, Floating 클래스에 대한 Behavior 인스턴스를 정의할 수 있다.

```
instance Num a => Num (Behavior a)
where
  (+) = lift2 (+)
  (*) = lift2 (*)
  negate = lift1 negate
  abs = lift1 abs
```

Floating에 정의되었던 함수들 또한 다음과 같은 overloading을 이용하여 lift된다.

```
instance Floating a
   => Floating (Behavior a) where
  pi    = lift0 pi
  sqrt  = lift1 sqrt
  exp   = lift1 exp
  log   = lift1 log
  sin   = lift1 sin
  cos   = lift1 cos
```

Lifting은 over에 대해서도 적용될 수 있다. 여기서, over외에 empty 함수가 소개되는 이유는 여러 앞서 Picture에서 Empty를 정의하는 이유와 같다. Combine에 대해서 여러 종류의 인스턴스들이 정의될 수 있다.

```
class Combine a where
   empty :: a
   over  :: a -> a -> a

instance Combine Picture where
  empty = EmptyPic
   over  = Over

instance Combine a
   => Combine (Behavior a) where
  empty = lift0 empty
  over  = lift2 over
```

overMany는 여러 Behavior Picture 들을 디스플레이 한다.

```
overMany :: Combine a => [a] -> a
overMany = foldr over empty
```

예를 들어, 노란색과 빨간색으로 깜박이면서 변화되면서 원형으로 움직이는 조그만 동그라미를 그리는 함수 flashingBall :: Behavior Picture 이 주어졌을 때, 이것을 8개의 동그라미가 깜박이면서 움직이는 것으로 확장시키는 것을 overMany를 이용하여 쉽게 정의할 수 있다.

```
revolvingBalls :: Behavior Picture
revolvingBalls
   = overMany [timeTrans (lift0 (t*pi/4) +
time) flashingBall | t <- [0..7]]
```

5. 결론 및 관련 연구

지금까지 순수 함수형 언어 Haskell을 이용한 애니메이션 프로그래밍 기법에 대해서 논의하였다. Haskell의 대수형 타입, 클래스 타입 및 인스턴스, 다형성, 고차함수, 모나드, 컴비네이터, Lifting 함수 정의 등은 함수형 프로그래밍 고유의 추상화 기능을 제공하고 있다. 원래 이러한 기능은 일반적인 모든 경우에 적용될 목적으로 개발되었는데, 본 논문에서 소개하는 바와 같이 그래픽과 애니메이션 프로그래밍에 매우 성공적으로 응용될 수 있음을 알 수 있다. 이 성공 요인은 다음과 같이 정리될 수 있다.

첫째, Hindley-Milner 타입에 의한 타입 변수의 사용과 타입 클래스는 polymorphic 함수와 overloaded 함수를 정의할 수 있도록 한다. 이러한 기능 덕택으로 프로그램을 좀

더 조직적으로 구성할 수 있으며, 더욱 다양한 다형성을 표현할 수 있도록 해 준다.

둘째, 고차함수 기능에 따라 프로그램 부분 (program fragments)을 매우 유연한 형태로 합성할 수 있도록 하며 액션들 또한 마치 일반 데이터처럼 프로그램에 의해 생성될 수 있도록 한다.

셋째, 위의 고차함수 기능과 더불어 프로그램의 합성을 원활하기 할 수 있는 여러 컴비네이터들을 정의할 수 있다. 기존에 널리 사용되었던 map, fold와 더불어 순차적 입출력을 가능케 하는 sequence_, 여러 타입의 내용을 한 화면에 중첩 표현하는 over는 애니메이션용 컴비네이터로서 매우 편리한 기능을 하고 있다. 일반적으로 대화형 특성을 갖는 프로그램의 합성은 어려운 문제이지만, 함수형 컴비네이터들은 이러한 문제를 완벽하게 처리하고 있다.

넷째, 대수형 타입은 매우 정교한 데이터 정의를 가능케 한다. Shape, Region, Picture와 더불어 이들 데이터 타입을 시간성을 포함하는 Behavior 타입으로 Lifting하는 기법을 구사함으로써 정교한 데이터 타입과 프로그램을 구성할 수 있도록 한다.

다섯째, CR 특성은 함수형 프로그래밍이 갖는 매우 근본적인 특성이다. 그러나, 이러한 특성을 유지하면서 동시에 입출력, 계산 순서의 순차화 등을 표현하는 것은 매우 어려운 일이었다. 최근 연구된 모나드 기법과 이를 기반으로 정의된 do 구문 덕택으로 이 두 가지를 모두 만족시킬 수 있게 되었다.

본 연구는 Hudak의 멀티미디어를 Haskell로 프로그래밍하는 기법 [6]과 Elliot에 의해서 연구되어 오고 있는 Fran [7][8]으로부터 많은 영향을 받았다. 특히 본 논문에서 소개되는 코드들의 많은 부분은 [6]에 소개된 것으로서 본 논문에서는 그 의미와 효과 등을 새롭게 조명하였다. [6]에는 그래픽 애니메이션 외에 음악 작곡 및 로봇 제어 등의 새롭고 독특한 프로그래밍 기법이 소개되어 있다. 이러한 분야 외에, 웹서버 개발 등 다양한 분야에 걸쳐 Haskell 응용 프로그래밍의 성공적인 구현 사례가 보고되고 있으므로 Haskell의 실용화에 대해서 기대를 갖게 한다.

감사의 글

이 연구는 경성대학교 멀티미디어 특성화사업 4차년도 연구비에 의하여 지원을 받았습니다.

참고 문헌

[1] Haskell home page. http://www.haskell.org/

[2] John Hughes. Why functional programming mattes. *Computer Journal*, 32(2):98-107, 1989.

[3] E Moggi. Computational lambda-calculus and Monads. In Proceedings of Symposium on Logic in Computer Science. IEEE. 1989.

[4] P. Wadler. Comprehending Monads. In Proceedings of Symposium on Lisp and Functional Programming, pp. 61-78. ACM. June 1990.

[5] Simon Thompson, Haskell: The Craft of Functional Programming. Addison-Wesley. 1996.

[6] P. Hudak. *The Haskell School of Expression: Learning Functional Programming Through Multimedia.* Cambridge University Press, March 2000.

[7] C. Elliott. "Composing Reactive Animations", Dr. Dobb's Journal, July 1998.

[8] C. Elliott and P. Hudak, "Functional Reactive Animation", June 1997.

변석우

1976~1980. 숭실대학교 전자계산(학사).
1980.~1982. 숭실대학교 전자계산(석사).
1982.~1999. ETRI 책임연구원
1988~1994. 영국 University of East Anglia 전산학(박사).
1998.~현재 경성대학교 정보과학부 조교수

관심분야는 rewriting system, 함수형 프로그래밍, 프로그래밍 언어, 정형 시스템 등.