Haskell에서의 패키지 데이터 형

장학상° 권기항
동아대학교 컴퓨터공학과
janghaksang@yahoo.co.kr°, khkwon@daunet.donga.ac.kr

Package Data Type in Haskell

Haksang Jang° Keehang Kwon
Dept. of Computer Engineering, Dong-A University

요 약

순수 함수형 프로그래밍 언어 Haskell에서의 데이터 추상은 대수 데이터 형과 관련 함수로 묶어진 모듈의 인터페이스만을 노출함으로써 이루어진다. 이러한 데이터 추상에 대한 제한된 용법은 프로그램 설계에 있어 유용하게 사용되는 패턴 매칭을 제약하고, 파일 단위의 추상을 요구함으로써 데이터 구조 단위의 데이터 추상을 불가능하게 한다. 본 논문은 Haskell의 제한된 데이터 추상 문제를 해결하기 위한 방법으로 패키지 데이터 형을 제안한다. 패키지 데이터 형은 더 작은 단위로 데이터 추상을 가능하게 하고, 패턴 매칭을 제약하지 않고 자유롭게 사용할 수 있게 한다.

1. 서 론

데이터 추상은 데이터 형의 성질과 그에 대한 구현을 분리하는 것이다.[1] 컴퓨터 언어에서 데이터 추상은 일반적으로 추상 데이터 형(abstract data type)을 사용함으로써 이루어진다.

순수 함수형 프로그래밍 언어 Haskell[2]에서의 추상 데이터 형은 대수 데이터 형(algebraic data type)과 관련 함수로 묶어진 모듈(module)의 인터페이스(interface)만을 노출함으로써 이루어진다. 대수 데이터 형의 구성자(constructor)는 이 인터페이스에 포함될 수도 그렇지 않을 수도 있다.

그러나 이러한 정책으로 인해 구성자를 인터페이스로 노출할 경우 다른 함수 정의에서 구성자가 사용되는 것을 막지 못하므로, 구체적인 구현을 숨기는 데이터 추상을 위배한다. 또한 구성자를 인터페이스에서 숨길 경우, Haskell이 가진 강력한 개념중 하나인 패턴 매칭(pattern matching)을 사용할 수 없게 한다.[3,4,5,6]

뿐만아니라 모듈 단위의 데이터 추상으로 인해 데이터 추상에 관련된 멤버 함수(member function)와 이러한 데이터 추상을 이용하는 함수에 대한 구분이 명확치 못하다.

본 논문은 Haskell의 제한된 데이터 추상 문제를 해결하기 위한 방법으로 패키지 데이터 형(package data type)을 제안한다. 패키지 데이터 형은 더 작은 단위로 데이터 추상을 가능하게 하고, 패턴 매칭을 제약하지 않고 자유롭게 사용할 수 있게 한다.

2. Haskell의 데이터 추상

Haskell은 데이터 추상을 사용하고 객체 형을 구현하기 위해 기본적으로 대수 데이터 형을 사용한다. 그림 1은 유리수의 형을 정의하고 있다.

```
data Rat = Rat { numer, denom :: Int }
```

그림 1 유리수의 대수 데이터 형

Rat은 Int 형 두 수를 매개변수로 취하고 유리수를 생성하는 구성자이며, Int→Int→Rat 형을 지닌다. numer와 denom은 매개변수로 유리수를 취하고 분자와 분모를 돌려주는 선택자(selector)이며, Rat→Int 형을 지닌다.

```
data Rat = Rat Int Int

rat :: Int → Int → Rat
numer, denom :: Rat → Int

rat x y | y ≠ 0 = Rat x y
numer (Rat x y) = x `div` (gcd x y)
denom (Rat x y) = y `div` (gcd x y)
```

그림 2 수정된 유리수의 대수 데이터 형

그러나 그림 1의 정의는 유리수를 제대로 표현하고 있지 못하다. 그림 2에서 정의된 함수 rat은 유리수가 만들어질때 분모가 0이 아닌지 검사한다. 또한 함수 numer와 denom은 유리수에 대한 분자와 분모의 비를 돌려줌으로써 유리수의 정의를 제대로 표현한다.

다만 그림 2는 구현 정보를 숨겨야 하는 데이터 추상 원칙을 위배한다.

```
module Rat (rat, numer, denom) where

data Rat = Rat Int Int

rat :: Int → Int → Rat
numer, denom :: Rat → Int

rat x y | y ≠ 0 = Rat x y
numer (Rat x y) = x `div` (gcd x y)
denom (Rat x y) = y `div` (gcd x y)
```

그림 3 구현에 대한 정보가 은닉된 유리수의 대수 데이터 형

Haskell에서는 데이터 형과 그에 대한 멤버 함수를 묶기 위해서 모듈을 사용하고 이에 대한 구현을 숨기기 위해 모듈에 대한 인터페이스를 정의한다. 그림 3은 완성된 형태의 유리수의 ADT(abstract data type)와 그에 대한 구현을 나타낸다.

그러나 이러한 형태의 ADT는 Haskell이 지닌 큰 장점중 하나인 패턴 매칭을 사용할 수 없게 만들고, 모듈 내에 위치한 데이터 형을 직접 다루기 위한 멤버 함수와 그 멤버 함수를 다루는 함수를 구별하기 어렵게 한다.

```
module Rat (rat, numer, denom, Rat (Rat)) where

data Rat = Rat Int Int

rat :: Int → Int → Rat
numer, denom :: Rat → Int

rat x y | y ≠ 0 = Rat x y
numer (Rat x y) = x `div` (gcd x y)
denom (Rat x y) = y `div` (gcd x y)
```

그림 4 패턴 매칭이 가능한 유리수의 대수 데이터 형

그림 4는 유리수의 구성자 Rat을 모듈 밖으로 노출함으로써 유리수의 ADT에 대해 패턴매칭이 가능하도록 노력하고 있다.
그러나 위의 정의는 패턴 매칭을 사용할 수는 있으나 Rat의 사용을 제약하지 못함으로 인해서 정보를 은닉할 수 없게된다.
결국 Haskell은 프로그램 설계에서 중요한 개념인 데이터 추상과 패턴 매칭을 서로 잘 조화하지 못한다[3,4,5,6].

3. 패키지 데이터 형

3.1 패키지 데이터 형의 제안

패키지 데이터 형은 대수 데이터 형 선언에 함수의 정의를 함께 지니고 있는 데이터 형이다. 데이터 형의 선언과 함수의 정의를 구분하고, 문맥을 통해 구성자의 사용에 제약을 가하기 위해 예약어 in을 사용한다. 예약어 in을 붙이지 않은 패키지 데이터 형은 모든 면에서 기존의 대수 데이터 형과 동일하다.
패키지 데이터 형에 정의되는 멤버 함수는 ADT를 정의하는데 필요한 구성자와 선택자, 조건자(predicate)가 가능하다. 이 세 종류의 함수는 리턴값이나 입력값의 형이 반드시 해당 함수를 포함하고 있는 패키지 데이터 형이다.

```
data Complex = Cart Float Float
    in
        cart :: Float → Float → Complex
        cartR :: Float → Complex
        real :: Complex → Float
        imag :: Complex → Float
        cart x y = Cart x y
        cartR x = Cart x 0.0
        real (Cart x y) = x
        imag (Cart x y) = y
```

그림 5 복소수의 패키지 데이터 형

그림 5는 복소수 형을 정의한다. 여기서 구성자 Cart는 Float →Float→Complex 형이다.
일반적으로 구성자는 두 가지 의미로 사용되는데, 오른쪽의 함수 정의 부분에 사용되는 구성자는 값을 조합하기 위한 역할로 사용되고, 왼쪽의 패턴 매칭 부분은 값을 분리하기 위해 사용된다.[2,4]

```
add :: Complex → Complex → Complex
add (Cart x y) (Cart x' y') = cart (x+x') (y+y')
```

그림 6 복소수의 덧셈

그림 6에서 구성자 Cart는 단지 왼쪽 패턴 매칭 부분에서만 사용되었다. 패키지 데이터 형 내에 구성자 함수 cart와 cartR이 정의되었기 때문에 원래의 구성자 Cart는 마치 객체지향 프로그래밍에서의 private 요소처럼 취급받는다. 즉, Cart는 오른쪽의 add내의 정의내에서는 사용할 수 없고, 단지 왼쪽에서 패턴 매칭을 위한 값의 분리를 위해서만 사용될 수 있다. 이러한 차이를 제외한다면 패키지 데이터 형은 모듈에서 정의할 수 있는 ADT의 의미와 동일하다.

3.2 Complex 패키지 데이터 형의 문제점

그러나 위의 복소수 정의는 복소수가 가진 여러 성질을 고려하면 불완전하게 된다. 복소수의 덧셈은 직각좌표를 사용해서 표현하는 것이 간단하지만, 곱셈의 경우에는 극좌표를 쓰는 것이 더 편하므로 복소수의 정의는 극좌표도 같이 표현할 수 있어야 한다. 직각좌표와 극좌표를 동시에 표현하는 가장 간단한 방법은 둘 다에 대한 구성자를 포함하는 것이다.

```
data Complex = Pole Float Float | Cart Float Float
    in
        cart, pole :: Float → Float → Complex
        cart x y = Cart x y
        pole r t = Pole r t
        ...

add, mult :: Complex → Complex → Complex
add (Cart x y) (Cart x' y') = cart (x+x') (y+y')
mult (Pole r t) (Pole r' t') = pole (r×r') (t+t')
```

그림 7 수정된 복소수의 정의와 복소수의 연산

그러나 그림 7에서는 식 (mult c1 c2)를 계산할때 패턴 매칭에서 오류가 발생할 수 있다. c1과 c2가 극좌표가 아닌 직각좌표로 되어있다면 mult를 계산할 수 없다. 이 문제에 대한 간단한 해결책으로 극좌표간의 내부적인 변환과 패턴 매칭을 위한 관련 함수를 일일이 만드는 방법을 사용할 수 있다. 그러나 이러한 방법은 결국 구현을 숨기기 힘들게 하고, 코드를 복잡하게 만들며, 가독성을 크게 떨어뜨린다. 결국, 여러 성질을 지닌 데이터는 패턴 매칭을 수행하기가 까다롭다.

3.3 Set 패키지 데이터 형의 문제점

유리수의 경우에는 패키지 데이터 형에서 구성자의 사용을 제약하고 패턴 매칭을 사용하는 것은 쓸모가 있었다. 이것은 데이터가 단일한 성질을 지니기 때문에 가능했다. 그러나 일반적으로 모든 경우에 유용하지는 않다.

```
module Set (Set, empty, isEmpty, member, insert, delete) where

data Set α = [α]
    in
        empty :: Set α
        isEmpty :: Set α → Bool
        member :: Set α → α → Bool
        insert :: Set α → α → Set α
        delete :: Set α → α → Set α
        empty = []
        isEmpty [] = True
        isEmpty s = False
        ...

-------------------------------------------------------------

module Set (Set, empty, isEmpty, member, insert, delete) where

data Set α = Null | Fork (Set α) α (Set α)
    in
        empty :: Set α
        isEmpty :: Set α → Bool
        member :: Set α → α → Bool
        insert :: Set α → α → Set α
        delete :: Set α → α → Set α
        empty = Null
        isEmpty Null = True
        isEmpty s = False
        ...
```

그림 8 리스트와 트리를 이용한 집합의 정의

그림 8의 집합은 단일한 성질을 지니지만 구현을 위해서 각각 리스트와 트리를 사용했다.[7] 이와같이 여러 형태의 구현이 있을 수 있는 데이터의 경우 구성자의 사용이 제약되었다고 하더라도 패턴 매칭으로 인해 그 구현을 드러나는 것을 피할 수 없고, 패턴 매칭을 수행하기가 까다롭다.

3.4 Queue 패키지 데이터 형의 문제점

형에 대한 여러 형태의 구현이 있을 때만 패턴 매칭을 사용하기 힘든 것은 아니다. ADT가 패턴 매칭을 사용하기 힘든 특별한 방법으로 구현 되어있을 경우에도 패턴 매칭을 사용하는 것이 적절치 못할 수 있다.

```
module Queue (Queue, empty, isEmpty, join, front, back) where

data Queue α = MkQ ([α],[α])
    in
        empty :: Queue α
        isEmpty :: Queue α → Bool
        join :: Queue α → α → Queue α
        front :: Queue α → α
        back :: Queue α → α
        mkValid :: ([α],[α]) → Queue α
        empty = MkQ ([],[])
        isEmpty (MkQ (xs,ys)) = null xs
        join (MkQ (xs,ys)) z = mkValid (xs, z:ys)
        front (MkQ (x:xs,ys)) = x
        back (MkQ (x:xs,ys)) = mkValid (xs,ys)
        mkValid (xs,ys) = if null xs then MkQ (reverse ys, [])
                          else MkQ (xs,ys)
```

그림 9 reverse를 이용한 큐의 정의

그림 9는 reverse를 사용해서 좌우 대칭인 형태로 큐를 설계했다.[7] 이렇게 설계된 Queue에 대한 패턴 매칭은 큐의 구현을 그대로 노출함으로써 데이터 추상 원칙을 위배한다.

3.5 패턴 매칭과의 결합

위에서 드러난 세 가지 문제점은 데이터 추상과 패턴 매칭간의 충돌에서 나타나는 문제가 아니라, 단지 패턴 매칭을 표현하는 방법에 대한 문제라고 볼 수 있다. 이 문제는 기존의 패턴 매칭에 대한 연구[3,4,5,6]를 통해 보완할 수 있다. 이러한 연구는 다양한 형태의 구현을 숨기면서 패턴 매칭에 대한 일관적인 인터페이스를 사용할 수 있게 한다.

기존의 패턴 매칭에 대한 연구 또한 패키지 데이터 형을 통해 보완할 수 있다. 예를 들어 view[3,5,6]는 패턴을 이용해서 구현을 숨기지만 구성자의 사용을 제약할 수는 없으나 패키지 데이터 형과 결합하여 더 안전한 프로그래밍을 할 수 있다. transformational pattern[4]은 패키지 데이터 형에 대해서 거의 중첩없이 결합해서 사용할 수 있다.

패키지 데이터 형은 데이터 추상을 향상하여 구성자에 대한 의미를 명시적으로 나타내고, 확장된 패턴 매칭에 의해 패턴 매칭의 다양한 표현 방법을 사용함으로써 데이터 추상과 패턴 매칭간의 충돌을 피할 수 있을 뿐만 아니라, 프로그램의 설계와 구현을 단순화하고 이해도를 향상할 수 있다.

4. 문법과 의미

패키지 데이터 형의 문법은 Haskell 98 Report[1]에 기반한다. 패키지 데이터 형은 그림 10에서 보는 바와 같이 data를 확장해서 만들어진다.

```
topdecl → ...
        | data [context ⇒] simpletype = constrs [deriving] [package]
package → in decls
```

그림 10 패키지 데이터 형의 문법

패키지 데이터 형이 선언되었을 때 문맥은 아래와 같은 규칙에 따라 변환할 수 있다.

1. 패키지 데이터 형을 사용하지 않은 경우
   - 변경 불필요
2. 패키지 데이터 형을 명시적으로 사용한 경우
   - data : decls를 모듈 상의 함수로 변경한다.
   - 멤버 함수 : 멤버 함수의 타입의 인자나 리턴값은 반드시 해당 멤버 함수를 포함하는 패키지 데이터 형이다.
   - 함수 : 함수의 오른쪽에 구성자가 올 수 없다.

```
case topdecl of [[
  data cx ⇒ T u1 ... uk = K1 t11 ... t1k1 | .. | Kn tn1 ... tnkn
      in
          f1 :: type1
          f1 a11 ... a1n = e1
          ...
          fn :: typen
          fn an1 ... ann = en

]] = [[
  data cx ⇒ T u1 ... uk = K1 t11 ... t1k1 | .. | Kn tn1 ... tnkn
  f1 :: type1
  f1 a11 ... a1n = e1
  ...
  fn :: typen
  fn an1 ... ann = en
]]
```

그림 11 패키지 데이터 형의 의미적 변환

6. 결론 및 향후 과제

본 논문에서는 Haskell의 제한된 데이터 추상 문제를 해결하기 위한 방법으로 패키지 데이터 형(package data type)을 제안했다. 패키지 데이터 형은 Haskell 언어에서 나타나는 패턴 매칭과 데이터 추상 간의 충돌을 없앤다. 뿐만 아니라 구성자의 의미에 자연스럽게 명시적으로 제약을 더함으로써 프로그래머가 데이터 추상이나 패턴 매칭에 관련된 문제에서 고민을 하지 않게 하고, 문제 해결을 위한 보다 다양한 사고를 도우며, 구현을 위한 효과적인 방법을 제공한다.

패키지 데이터 형은 현재의 Haskell 언어에서 최소한의 확장이다. 이러한 점은 문법적인 문제 뿐만 아니라 구현의 문제를 매우 단순하게 만든다.

선언된 패키지 데이터 형은 작은 모듈로 인식될 수 있기때문에 파일단위의 추상이 아닌 데이터 단위의 추상이 가능하다. 이는 Haskell이 지닌 객체지향적 특성을 한층 더 끌어올리고, 프로그램 설계에서의 모듈 관리에 대한 부담을 줄일 수 있는 요소가 될 수 있다.

다만 현재 데이터 추상을 위해 확장된 패턴 매칭은 구현된 바가 없다. 때문에 패키지 데이터 형과 확장된 패턴 매칭간의 결합에 대한 엄밀한 연구를 위해 구체적인 사례를 연구하고 적절한 기법을 도출해낼 필요가 있다.

참고문헌

[1] H. Abelson and G. J. Sussman, The Structure and Interpretation of Computer Programs, MIT Press, Cambridge, MA, 1985.

[2] S. L. Peyton Jones, J. Hughes et al, Report on the Programming Language Haskell 98. 1999.

[3] P. Wadler, Views : A Way for Pattern Matching to Cohabit with Data Abstraction. In 14th ACM Symp. on Principles of Programming Languages, pp.307-313, 1987

[4] M. Erwig and S. P. Jones, Pattern Guards And Transformational Patterns, Electronic Notes in Computer Science, pp12.1-12.27, 2001.

[5] P. P. Gostanza, R. Pena and M. Nunez, A New Look at Pattern Matching in Abstract Data Types, In 1st ACM SIGPLAN Int. Conf. on Functional Programming, pp110-121, 1996

[6] F.W. Burton, E. Meijer, P. Sansom, S. Thompson, and P. Wadler, Views: An Extension to Haskell Pattern Matching, http://haskell.org/development/views.html, 1996

[7] R. Bird, Introduction to Functional Programming using Haskell, Prentice Hall Europe, 1998