귀납적 자료형의 이진화를 이용한 타입 레벨 프로그래밍 간소화

차리서[0] 최진영
고려대학교 컴퓨터학과
{reeseo[0], choi}@formal.korea.ac.kr

Simplifying Type-level Programming by Booleanizing Inductive Types

Reeseo Cha[0] Jinyoung Choi
Dept. of Computer Science and Engineering, Korea university

요 약

Dependent type은 유리수, 리스트 함수, 행렬 곱 등 여러 가지 타입들의 제약 사항들을 충실히 표현하는 데에 필수적이기 때문에 이를 지원하는 타입 시스템을 탑재한 언어를 새로 개발하거나 기존 언어의 다른 특성들을 활용하여 이를 모의(simulate)하려는 시도가 다각도로 진행되고있으며, Haskell 타입 레벨 프로그래밍도 이런 모의 기법 중 하나다.

기존 타입 레벨 프로그래밍은 변별력의 손실이 없는 대신 이로 인해 관련 함수들의 타입이 복잡해지거나 확정하기 어려워지는 경우가 많아서 잘못된 프로그램을 작성할 위험 부담이 커진다. 실제로 dependent type이 필요한 경우들 중에는 매우 간단한 변별력만을 요구하는 경우가 많으므로, 귀납적 자료형을 이진 추상화하여 일부 변별력을 포기하는 대신 상대적으로 간단하게 dependent type과 관련 함수들의 타입을 확정하는 간소화된 타입 레벨 프로그래밍 기법을 제안한다.

1. 서론

소프트웨어에 대해 소스 코드 수준의 안전성을 확보하기 위한 여러가지 노력의 일환으로서, 기 작성된 코드에 대한 정적 분석(static analysis)이나 기 검증된 설계 명세로부터 안전한 방법으로 소스 코드를 추출해내는 코드 생성 (code generation) 기법 등과 함께 안전한 (safe) 프로그래밍 언어를 개발하려는 연구도 진행되고 있다. 근래 대다수의 언어에 적용된 컴파일 타임 구문 검사 (syntax checking) 기능은 안전한 언어를 개발하려는 연구의 대표적 성과물이었고, 이후 프로그램의 의미(semantics)까지 컴파일 타임에 검사하려는 시도가 이루어지고 있다.

이 중 가장 두드러진 연구 분야는 형-안전한 (type-safe) 프로그래밍 언어로서, 최소한 해당 언어에 탑재된 타입 시스템이 표현할 수 있는 범위의 의미에 대해서는 컴파일 타임에 정형적으로 검증하는 일이 가능해졌다.[1] 다만 현재의 타입 시스템은 아직 그 표현력이 완전하지 않아서, 종종 올바른 의미의 계산인데도 그 타입을 통해 올바름을 확인하지 못하거나 옳지 않은 의미의 계산임에도 그 계산을 컴파일 타임에 거부하지 않는 경우가 여전히 남아있다.

이에, 기존 타입 시스템들의 표현력을 강화하여 보다 많은 의미론적 제약 조건들을 컴파일 타임에 감지하고 검증해내는 방법은 최근 형-안전한 프로그래밍 언어 진영의 주요 관심사 중 하나가 되었다.

2. 기존 연구와 문제점

2.1 Dependent Types

형 체계의 표현력 확장에 관한 최근의 이슈 중 하나는 dependent type이다.[2] Dependent type이란 값에 의존하는 자료형이며, polymorphic type의 생성자가 매개변수로서 형 변수(type variable)만을 취할 수 있는 것과 달리[3] dependent type의 생성자는 값 수준 변수(term variable)를 취할 수 있다.

Haskell[4] 언어는 ML[5] 등과 함께 널리 알려져 쓰이고있는 형-안전한 범용 함수형 언어[6] 중에서는 상대적으로 단연 강력한 형 체계를 탑재한 언어 중 하나다. 그러나 현재 표준인 Haskell 98 명세는 dependent type을 지원하지 않으며, 따라서 Haskell에서는 간혹 컴파일 타임의 형 검사만으로는 프로그램의 중요한 의미를 검증하지 못하는 경우가 있다. 예를 들어, 유리수 형은 분모가 0일 수 없다든가 행렬은 대응되는 차원의 행렬끼리만 곱해야한다는 제약 조건들은 Haskell 형 체계로 충분히 표현할 수 없어서, 분모가 0인 유리수나 차원이 대

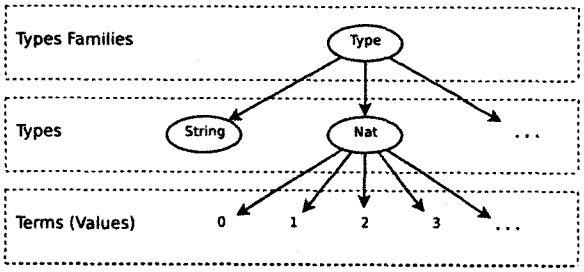

그림 1 정상적인 Haskell 프로그래밍

응되지 않는 행렬끼리의 곱은 런타임에 가서야 그 계산이 잘못되었음을 감지하게 된다.

새로운 프로그래밍 언어에 dependent type을 표현하고 검증할 수 있는 형 체계를 탑재하거나 기존 언어로 이에 준하는 효과를 얻으려는 노력이 최근 다양하게 진행되고 있다. Dependent type을 형 체계에 포함시킨 새로운 언어로는 Haskell의 변종인 Cayenne[7]이나 ML의 변종인 Dependent ML, 그 외에도 Epigram, Xanadu 등 여러가지가 있으며, 기존의 (자체적으로 dependent type을 지원하지 않는) 형-안전한 언어를 통해 dependent type의 효과를 모의(simulate)하려는 노력 중에는 타입 레벨 프로그래밍이 있다.

2.2 타입 레벨 프로그래밍 (TLP)

Conor McBride의 "Faking It"으로 대표되는 타입 레벨 프로그래밍 (TLP) 기법은 Haskell 형 체계가 제공하는 타입 클래스를 통해서 값 (텀) 수준의 계산 대상을 형 수준으로 끌어올려 표현하는 방식이다.[8] 예를 들어, 기존 자연수 타입은 (셀 수 있을 만큼) 그림 1처럼 무한한 값들을 원소로 갖지만, TLP의 자연수 타입은 실제로는 (셀 수 있을 만큼) 무한한 단원소 (singleton) 타입들을 원소로 갖는 타입 클래스다. 즉, 자연수 타입 '클래스'에 속하는 실제 타입인 0 타입, 1 타입, 2 타입 등은 각각 실제 값인 0, 1, 2 등과 일대일 대응을 이룸으로써, 계산 대상인 값(텀)을 대변하여 마치 값인 듯이 쓰일 수 있는 타입이 된다. (그림 2)

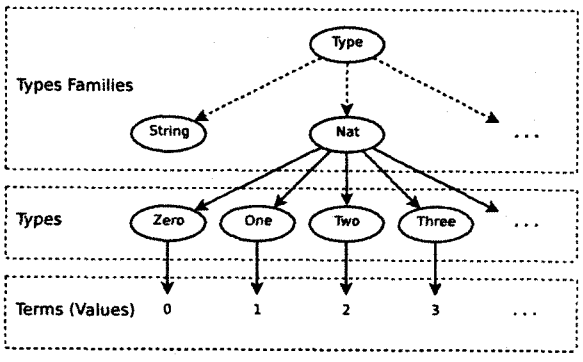

그림 2 기존 타입 레벨 프로그래밍

이렇게 값을 타입 수준으로, 다시 타입을 타입 클래스 수준으로 각각 끌어올림으로써, 모든 형 생성자는 형에만 의존해야 하는 Haskell 형 체계의 제약 조건을 만족시키는 채로 dependent type 개념이 요구하는 '값에 의존하는 자료형'을 모의할 수 있게된다.

2.3 기존 TLP의 문제점

Dependent type을 모의하는 이런 방식은 값 변별력에 아무런 손실이 일어나지 않으므로 모든 dependent type에 공통적으로 적용할 수 있는 장점이 있지만, 반면에 이런 강력한 변별력 때문에 모의된 dependent type에 관한 함수 형이 복잡해지는 경우가 있다. 예를 들어 정상적인 Haskell 프로그래밍에서 두 자연수에 대한 덧셈 함수 add의 타입은 간단히

```
add :: (Num a) => a -> a -> a
```

로 확정할 수 있는 반면, 기존 TLP에서 두 자연수 계열 타입들에 대한 덧셈 함수 add의 타입은 인수의 값, 즉 실제로는 인수 값에 일대일로 대응되는 타입에 따라 (셀 수 있을 만큼) 무한히 다른 종류의 타입을 갖게 된다. 물론 GHC나 Hugs의 확장 기능인 functional dependency를 활용하여 이 문제를 해결하는 편법이 있지만 그 과정은 여전히 지나치게 복잡하다.

만일 행렬 곱의 경우처럼 모든 값들끼리 서로 구별할 수 있는 완전한 변별력이 필요하다면 다른 대안이 없으므로 이런 복잡한 과정을 거쳐야겠지만, 때로는 이런 변별력이 다 필요하지 않은 경우도 있다. 예를 들어 유리수 타입은 분모에 해당하는 값이 0인지 0이 아닌지만 구별할 수 있으면 되고, 리스트에 대한 head나 tail 함수는 리스트가 빈 리스트인지 아닌지만 구별할 수 있으면 언제나 안전하다. 이런 경우에 한해서는 굳이 불필요한 변별력을 유지하여 함수의 타입을 복잡하게 만들 필요가 없으며, 정확히 필요한 만큼의 변별력만 남김으로써 이런 dependent type에 관한 함수의 타입도 간단히 확정해주는 방법을 고려할 필요가 있다.

3. 타입 이진화를 통한 TLP 간소화

3.1 타입 이진화

유리수나 리스트 함수의 경우에서 볼 수 있듯이, 많은 경우에 실제로 필요한 변별력은 결국 '경계값이냐 아니냐'라는 이진 평가 문제로 귀결된다. 특히 자연수나 리스트처럼 귀납적으로 정의된 자료형의 경우에는 귀납 기초에 대응하는 값인지 귀납 단계가 적용된 값인지만을 서로 구별할 필요가 있을 뿐이다. 이런 이진화 개념을 타입 레벨 프로그래밍에 적용하면 문제를 간소화시킬 수 있다.

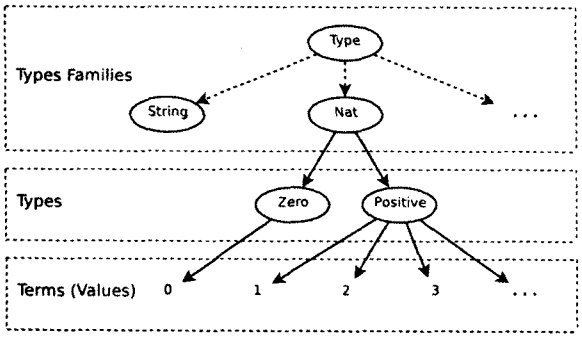

그림 3 타입 수준 이진화 TLP

자연수의 경우를 예로 들면, 많은 경우에 우리가 필요로 하는 변별력은 해당 자연수 값이 0이냐 양수냐의 문제일 뿐이며, 그림 3과 같이 기존 TLP에서 자연수 타입 클래스에 속했던 무한한 수의 타입들을 0 타입과 양수 타입 두 가지만으로 줄일 수 있다.

$$\frac{}{\Delta \vdash \mathbf{Nat'_B} : 2^{\mathbf{Type}}}$$

$$\frac{}{\Delta \vdash \mathbf{Zero'_B} : Zero'_B} \qquad \frac{\Delta,\ t : \mathbf{Nat'_B} \vdash n : t}{\Delta \vdash \mathbf{Succ'_B}\ n : Pos'_B}$$

$$\frac{}{\Delta \vdash Zero'_B : \mathbf{Nat'_B}} \qquad \frac{}{\Delta \vdash Pos'_B\ t : \mathbf{Nat'_B}}$$

수식 1 기본적인 타입 이진화 규칙

이에 대응하는 기본적인 자연수 타입 생성 규칙은 수식 1과 같다. 먼저 자연수 계열 $\mathrm{Nat_B}$'를 타입 클래스로 선언하고 0 타입을 나타내는 싱글톤 타입 $\mathrm{Zero_B}$'을 선언한다. 여기에 모든 자연수 계열 타입의 원소에 대한 계승값은 양수 타입이 되도록 양수 타입의 소개 규칙을 만들고, 마지막으로 0 타입과 양수 타입이 모두 자연수 계열 타입들임을 명시한다.

그러나 이런 방식으로 만든 타입 소개 규칙은 Haskell 타입 클래스의 특성상 곧바로 구현하기 곤란하므로, Haskell 타입 선언 규칙에 부합하도록 양수 타입 계열에 대한 귀납 기초를 명시적으로 분리해주어 수식 2와 같이 변형된 타입 생성 규칙을 얻는다. 즉, 다시 귀납적으로 정의되는 양수 타입은 1 (One)을 귀납 기초로 하여 모든 양수 p의 계승이 양수가 되도록 정의한다.

또한, 수식 2의 $\mathrm{Nat_B}$의 경우 임의의 양수 p로부터 새로운

$$\frac{}{\Delta \vdash \mathbf{Nat_B} : 2^{\mathbf{Type}}} \qquad \frac{}{\Delta \vdash \mathrm{Zero_B} : Zero_B}$$

$$\frac{}{\Delta \vdash \mathrm{One_B} : Pos_B} \qquad \frac{\Delta \vdash p : Pos_B}{\Delta \vdash \mathrm{Succ_B}\ p : Pos_B}$$

$$\frac{}{\Delta \vdash Zero_B : \mathbf{Nat_B}} \qquad \frac{}{\Delta \vdash Pos_B\ t : \mathbf{Nat_B}}$$

수식 2 양수에 대한 귀납기초 단계 분리

양수 p+1을 얻는 방법은 항상 공통이지만, $\mathrm{Nat_B}$'의 경우와는 달리 이 공통적인 방법을 0에 적용하여 1을 얻을 수는 없다. 이 문제를 해결하여 모든 자연수로부터 그 다음 자연수를 생성하는 공통적인 방법을 만들기 위해서, 상기 타입 생성 규칙에 최종적으로 수식 3 의 규칙을 추가한다.

$$\frac{\Delta \vdash t : \mathbf{Nat_B}}{\Delta \vdash succ_B : t \rightarrow Pos_B}$$

수식 3 공통 메서드 추가

이는 (Haskell의 관점에서 보면) 어떤 타입이 자연수 계열에 속하기 위해서는 반드시 그 타입의 원소에 공통 메서드 succ을 적용하여 양수를 얻을 수 있도록 succ이 정의되어있어야 한다는 뜻이며, 이상의 타입 소개 규칙들은 다음과 같은 Haskell 코드로 구현된다.

```
data ZeroType = O
data PositiveType = One | S PositiveType

class Nat t where
        s :: t -> PositiveType
instance Nat ZeroType where
        s x = One
instance Nat PositiveType where
        s x = S x
```

3.2 모의 dependent type에 대한 함수 타입

이와 같이 표현한 dependent type에 대한 함수는 기존 TLP에서 만들어내는 dependent type에 대한 함수에 비해 타이핑이 간단해진다. 예를 들어 두 자연수를 더하는 함수 add는 기존 TLP에서 인수의 값 따라 무한한 종류의 타입을 가질 수 있었지만, 이진 추상화된 TLP에서는 반드시 네 가지 타입 중 하나로 확정된다: {Z→Z→Z, Z→P→P, P→Z→P, P→P→P}

또한, 이 add 함수는 GHC/Hugs 확장 기능으로 제공되는 functional dependency를 통해 매우 간단하게 구현된다.

```
class Add a b c | a b -> c where
        add :: a -> b -> c
instance Add ZeroType ZeroType ZeroType where
        add O O           = O
instance Add ZeroType PositiveType PositiveType where
        add O One         = One
        add O (S y)       = S y
instance Add PositiveType ZeroType PositiveType where
        add One  O        = One
        add (S y) O       = S y
instance Add PositiveType PositiveType PositiveType
where
        add One One       = S One
        add One (S y)     = S (S y)
        add (S x) One     = S (S x)
        add (S x) (S y)   = S (S (add x y))
```

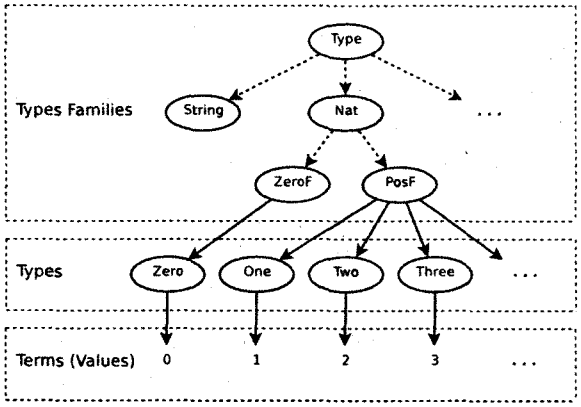

그림 4 타입 계열 수준에서의 이진화

4. 타입 클래스 이진화를 통한 변형된 간소화 기법

앞 절에서 제안했던 dependent type 모의 방법(Nat_B)은 대응 함수들의 타입을 비약적으로 간소화시키지만, 그 반대급부로서 상당한 변별력 손실을 감수할 수 밖에 없기 때문에 유리수 등 일부 경우에만 적용될 수 있는 문제가 있다. 이에, 이진화 기법을 TLP에 적용하여 문제를 단순화시키는 기본 골격을 유지하면서도 필요한 경우 충분한 변별력을 발휘할 수 있도록하는 또다른 변종을 제안한다. 이 변종 기법의 핵심은 이진화를 타입 수준에서 수행하는 대신 타입 클래스 수준에서 수행함으로써 타입 수준의 변별력을 그대로 유지하는 것이다.

앞에서와 마찬가지로 자연수의 경우를 예로 들면 그림 4와 같이 도식화되며, 이 기법에 대응하는 타입 생성 규칙은 수식 4와 같이 확정된다.

다만, 이런 타입 생성 규칙을 현재 Haskell에서 직접 구현하면 Nat_B의 경우와는 달리 add 등의 함수 타입이 다시 복잡해지게 되며, 향후 해결해야할 과제이다.

5. 결론

기존 TLP 기법은 손실없는 변별력으로 완전한 dependent type을 모의할 수 있는 반면, 경우에 따라서는 관련 함수 등의 타입이 불필요하게 복잡해지는 문제가 있다. 여기에 일종의 추상화(abstraction)로서 타입 이진화 기법을 적용하면 (Nat_B) 변별력이 손실되어 행렬 곱 등의 문제에는 활용할 수 없게되지만, 유리수나 리스트 관련 함수 등 경우에 따라서는 정확히 필요한 변별력만을 유지함으로써 훨씬 단순하게 관련 함수들의 타입을 확정할 수 있게된다. 더 나아가 Nat_S의 경우처럼 변별력을 기존 TLP 만큼 유지한 채로 타입 클래스 수준에서 이진 추상화를 시도할 수도 있지만, 이 경우에는 직접 Haskell 코드로 대응시키기 어려워지는 문제가 남아있다.

$$\frac{}{\Delta \vdash \mathbf{Nat_S} : 2^{\mathbf{Type}}}$$

$$\frac{}{\Delta \vdash \mathsf{Zero_S} : \mathit{Zero_S}} \qquad \frac{\Delta,\ t : \mathbf{Nat_S} \vdash n : t}{\Delta \vdash \mathsf{Succ_S}\ n : \mathit{Succ_S}\ t}$$

$$\frac{}{\Delta \vdash \mathit{Zero_S} : \mathbf{Nat_S}} \qquad \frac{\Delta \vdash t : \mathbf{Nat_S}}{\Delta \vdash \mathit{Succ_S}\ t : \mathbf{Nat_S}}$$

$$\frac{}{\Delta \vdash \mathbf{NatZero_S} : 2^{\mathbf{Nat_S}}} \qquad \frac{}{\Delta \vdash \mathbf{NatPos_S} : 2^{\mathbf{Nat_S}}}$$

$$\frac{}{\Delta \vdash \mathit{Zero_S} : \mathbf{NatZero_S}} \qquad \frac{\Delta \vdash t : \mathbf{Nat_S}}{\Delta \vdash \mathit{Succ_S}\ t : \mathbf{NatPos_S}}$$

수식 4 타입 계열 이진화 규칙

본 논문에서 제안한 이진 추상화 기법은 TLP에 적용하여 dependent type 모의를 간소화하는 방식 뿐만 아니라 독립된 형 체계를 개발하는 데에도 적용할 수 있으리라 보고 추가적인 연구를 진행하고 있다. 더불어 TLP 기법으로 얻은 dependent type을 활용하여 부분 함수 (partial function) 등[9] 더 많은 개념을 Haskell과 같은 기존 언어로 표현하고 검증하는 문제도 함께 고려하고 있다. 또한, 본 논문에서 마지막으로 제안했던 변종 Nat_S를 Haskell에서 실제로 구현하는 기법에 관한 연구도 아울러 진행중이다.

A. 참고 문헌

[1] David Turner, Philip Wadler, and Christian Mossin, "Once upon a type", ACM FPCA '95: 1－11, 1995.

[2] James McKinna, "Why dependent types matter", ACM SIGPLAN-SIGACT on POPL '06: 1－1, 2006.

[3] Keith Wansbrough and Simon Peyton Jones, "Once upon a polymorphic type", ACM SIGPLAN-SIGACT on POPL '99: 15－28, 1999.

[4] Paul Hudak and Joseph H. Fasel, "A gentle introduction to Haskell", SIGPLAN Not., 27(5):1－52, 1992.

[5] Robin Milner and David MacQueen, *The Definition of Standard ML*, MIT Press, 1997.

[6] John Hughes, "Why Functional Programming Matters", Computer Journal, 32(2):98－107, 1989.

[7] Lennart Augustsson, "Cayenne—A Language with Dependent Types", ACM SIGPLAN on ICFP '98: 239－250, 1998.

[8] Conor McBride, "Faking It—Simulating Dependent Types in Haskell" J. Funct. Program., 12(5):375－392, 2002.

[9] Simon Finn, Michael Fourman, and John Longley, "Partial Functions in a Total Setting", J. Autom. Reason., 18(1):85－104, 1997.