지연 함수형 프로그래밍 언어의 항 개서 의미
(Term Rewriting Semantics of Lazy Functional Programming Languages)

변 석 우 †
(Sugwoo Byun)

요 약 대부분의 함수형 프로그래밍 언어에서는 '위에서 아래쪽, 왼쪽에서 오른쪽 방향으로' 패턴 매칭(pattern matching)을 한다는 전략에 따라, *모호한(ambiguous)* 특성을 갖는 룰의 정의를 허용하고 있다. 이 방법은 함수형 프로그래머에게 디폴트 룰을 정의할 수 있게 하는 직관적인 편리함을 제공하지만, 한편으로 모호한 룰 때문에 함수형 언어의 의미는 불명확해 질 수 있다. 좀 더 구체적으로, 함수형 언어가 갖는 대표적인 특성인 등식 추론(equational reasoning) 원리의 적용을 불가능하게 할 수 있으며, 함수형 언어를 람다 계산법으로 변환하는 데 있어서도 정형적인 방법이 아닌 임시방편적인*(ad hoc)* 방법에 의존할 수 밖에 없게 한다. 본 연구에서는 *지연(lazy)* 함수형 언어의 패턴 매칭의 의미를 순수 선언적 특성을 갖는 항 개서 시스템(Term Rewriting Systems)의 *분리성(separability)* 이론과 연관시키고, 분리성 이론에 따라 지연 함수형 언어가 람다 계산법으로 변환될 수 있음을 보인다.

키워드 : 지연 함수형 언어, 항 개서 시스템, 패턴 매칭, 분리성, 람다 계산법

Abstract Most functional programming languages allows programmers to write *ambiguous* rules, under the strategy that pattern-matching will be performed in a direction of 'from top to bottom' way. While providing programmers with convenience and intuitive understanding of defining default rules, such ambiguous rules may make the semantics of functional languages unclear. More specifically, it may fail to apply the equational reasoning, one of most significant advantage of functional programming, and may cause to obscure finding a formal way of translating functional languages into the λ-calculus; as a result, we only get an *ad hoc* translation. In this paper, we associate with *separability* of term rewriting systems, holding purely-declarative property, pattern-matching semantics of *lazy* functional languages. Separability can serve a formalism for translating lazy functional languages into the λ-calculus.

Key words : Lazy Functional Languages, Term Rewriting Systems, Pattern Matching, Separability, Lambda Calculus

1. 서 론

대부분의 최신 함수형 프로그래밍 언어는 항 개서 시스템(TRS, Term Rewriting Systems)나 람다 계산법(Lambda Calculus)의 원리를 따르고 있으므로 이들 시스템의 강력한 이론적 기반을 적용할 수 있는 장점을 가지고 있다. 함수형 언어는 특히 룰(rules)의 정의 및 계산에서 패턴 매칭(pattern matching)을 이용하고 있으므로 자연스럽게 TRS와의 연관될 수 있다. 그러나, 선언적(declarative) 특징을 갖는 TRS와는 달리, 함수형 언어는 '위에서 아래쪽, 왼쪽에서 오른쪽 방향으로'의 패턴 매칭을 적용한다는 원칙 하에 아래와 같이 *모호한 (ambiguous)* 룰의 정의를 허락하고 있다.

예 1. (Haskell로 표현된 Factorial 함수)

```
fac 0 = 1
fac n = n * fac (n-1)
```

위에 정의된 Factorial 함수를 TRS의 측면에서 보면, *텀(term)* `fac 0`가 첫 번째와 두 번째 룰이 모두 매치

· 이 논문은 2005학년도 경성대학교 학술지원연구비에 의해 연구되었음

† 정 회 원 : 경성대학교 컴퓨터정보학부 교수
swbyun@ks.ac.kr

논문접수 : 2007년 12월 11일
심사완료 : 2007년 12월 28일

정보과학회논문지 : 시스템 및 이론 제35권 제3호(2008.4)

되므로 위의 룰은 모호하다. 모호한 룰은 TRS의 매우 중요한 특성인 CR(Church-Rosser, 혹은 '수렴'이라고도 함)을 만족시키지 못한다. 그러나 함수형 언어의 '위에서 아래' 방향의 패턴 매칭 때문에 `fac 0`은 언제나 첫 번째 룰과 매칭되고 두 번째 룰과 매칭되지 않으므로, 실제로는 모호한 현상이 발생되지 않는다. `fac`의 두 번째 룰은 0을 제외한 모든 정수를 매칭하는 디폴트(default) 룰의 역할을 한다. 이와 같이, 함수형 언어는 독특한 패턴 매칭 전략을 이용하고 있으며, 사용자는 이 전략을 쉽고 직관적으로 이해하면서 디폴트 룰을 정의할 수 있는 편리함을 갖는다.

그러나, 이러한 패턴 매칭 전략때문에 함수형 언어의 의미를 이해하는 것이 어렵게 되는 경우가 있다. [1]에 소개된 Haskell로 작성된 다음과 같은 프로그램을 고려해 보자.

예 2. (Haskell 프로그램 예)

```
data List          = Nil | Cons Int List
h Nil Nil          = 1
h Nil (Cons x y)   = 2
h (Cons x y) z     = 3
g Nil Nil          = 1
g (Cons x y) Nil   = 2
g z (Cons x y)     = 3
c x y z            = x z y
loop               = loop
```

이 프로그램에서 `loop`는 종료되지 않으며, `h`와 `g` 함수는 단지 두 인수의 순서만 서로 바꾼 것임을 알 수 있다. 그리고 `c` 함수는 어떤 이진 함수의 인수의 순서를 바꾸는 기능을 한다. 이 룰을 TRS의 관점에서 보면 선언적 특성에 따라 `c g = h`가 성립하지만, Haskell의 경우는 다음과 같은 문제점을 갖는다.

텀 `h(Cons 1 Nil) loop`를 룰 `h`와 패턴매칭시킬 때 첫 번째 인수인 `(Cons 1 Nil)`은 정규형(normal form) 이지만 `loop`는 레덱스(redex)이다. 이 텀은 `h`룰의 세 번째 룰과 칭되는데, 이때 `loop`는 `h` 룰의 변수 `z`와 매칭되므로 지연계산(lazy evaluation)의 원리에 따라 축약(reduction) 될 필요가 없고, 따라서 3으로 축약된다. 텀 `g loop(Cons 1 Nil)`는 세 번째 룰과 매칭될 수 있으나, '위에서 아래' 방향의 패턴 매칭 전략에 따라, 그 전에 레덱스 `loop`를 축약시켜 그 결과가 두 번째 룰의 `(Cons 1 Nil)`와 매칭되는지를 점검하는 과정이 선행된다. 그러면 `loop`가 정규형을 갖지 못하므로 텀 `g loop(Cons 1 Nil)`의 계산은 성공할 수 없게 된다. 따라서 Haskell에서는 TRS와는 달리 `c g = h`의 등식 추론이 성립되지 않는다.

예 2는 Haskell의 패턴 매칭 전략이 사용자가 기대하는 등식 추론의 원리를 방해하고, 함수형 언어와 TRS 사이의 의미가 서로 다름을 보여 주고 있다. 따라서 TRS에서 잘 정립된 이론적 환경을 충분히 이용할 수 없으며, 함수형 언어의 의미를 람다 계산법으로 해석하는 것 또한 어려워지게 된다. 지금까지 임시 방편적으로 (*ad hoc*) 함수형 언어를 람다 계산법으로 연관시키는 예는 많았음에도 불구하고(예를 들어, [2]) 이 둘의 연관성을 이론적 근거 하에 정형적으로 해석하는 예는 없었던 것으로 알고 있다. 이를 위해서는 먼저 함수형 언어의 의미를 좀 더 명확히 이해하는 것이 선행되어야 한다.

본 연구에서는 Haskell과 같은 *지연(lazy)* 함수형 언어의 패턴 매칭 의미를 순수 선언적 특성을 갖는 TRS의 *분리성(separability)* 이론과 연관시켜 설명한다. 이 연구의 결과는 지연 함수형 언어를 람다 계산법으로 변환하는 데 필요한 이론적 기반으로 이용될 수 있다.

이 논문에서는 제2장에서 기초 정의 및 TRS의 개념을 간략히 제공하고, 제3장에서 함수형 프로그램 언어를 분리성을 갖는 TRS로 변형시키는 알고리즘을 소개함으로써 함수형 프로그래밍 언어가 TRS의 분리성을 갖고 있음을 보인다. 이 분리성을 기반으로 지연 함수형 언어가 람다 계산법으로 해석될 수 있음을 설명한다. 제4장에서는 관련 연구를 소개하고, 본 연구의 의미를 정리한다.

본 연구에서의 논의는 독자들이 함수형 프로그래밍과 TRS 및 람다 계산법에 기본적 지식을 가지고 있음을 전제로 하고 있다. Haskell 프로그래밍에 구체적인 관심이 있는 독자는 [3], TRS에 대한 자세한 내용은 [4], 람다 계산법에 대해서는 [5]를 참조하기 바란다.

2. TRS의 기초 개념 및 정의

2.1 정의

함수 기호들의 집합과 변수들의 집합이 주어졌을 때, 모든 변수들은 텀(term)이 될 수 있다. 또한, n개의 인수를 갖는 함수 기호 F와 n개의 텀 $t_1, \ldots, t_n$ $(n \geq 0)$이 주어졌을 때 $F(t_1, \ldots, t_n)$ 또한 텀이다. 한 텀의 특정 위치는 자연수들의 순열(sequence)로 구성된 *주소(address)*로서 표현된다. 한 텀 $t \equiv F(t_1, \ldots, t_n)$와 주소 $u \equiv i \cdot v$가 주어졌을 때(여기서 i는 자연수이며 v는 주소이다.) t의 *부분 텀(subterm)* $t|u$는 $t_i|v$가 되며 $<>$는 빈 주소를 의미한다. 즉, $t|<> = t$. 한 텀 t의 주소들의 집합은 $O(t)$로서 표현한다. u가 텀 t의 부분 텀의 주소인 경우, $t[u:=t']$은 u 위치에 있는 t의 부분 텀을 t'으로 대치한 결과를 의미한다. $t|u$가 변수가 아닌 경우 $t|u$는 *진 부분 텀 (proper subterms)*이라 일컫는다.

변수의 *치환(substitution)*은 유한 개의 변수들로부터 텀들로 사상하는 함수 σ로서 표현된다. $\sigma(t)$는 텀 t에 있는 변수 x를 $\sigma(x)$로 대치한 결과를 나타낸다. t'이 t의

치환 실례화(substitution instance)인 경우 $t \leq_s t'$로서 표시한다. t와 t'에 대해서 $t \leq_s t'' \geq_s t'$으로 되는 어떤 t''이 존재하면 t와 t'는 동일화가 (unify)될 수 있으며 $t \uparrow_s t'$으로 표현한다. 어떤 한 텀 t가 주소 u에 있는 텀 t'의 부분 텀과 *매칭 (matching)* 되기 위해서는 $\sigma(t) = t'|u$이 성립해야 한다. 이때, $v = u \cdot r$ 형태로 된 t'의 주소 v에 대해서 r이 t의 주소이고, $t|r$가 진 부분 텀일 때 t'의 v는 매칭되었다고 일컫는다.

TRS는 (Σ, $\mathcal{R}$, T)로서 구성되어 있다. 여기서 Σ는 함수 기호들의 집합이고, $\mathcal{R}$는 개서 룰(rewrite rules)들의 집합이며, T는 Σ로 구성되는 텀들의 집합이다. 룰은 두 개의 텀 t_1, t_2가 주어졌을 때, $t_1 \rightarrow t_2$의 형태로 표현된다. 보통 Σ와 T가 고정되어 있다는 가정 하에 TRS를 $\mathcal{R}$로 표현한다. $l\mathcal{R}$은 룰의 왼쪽에 위치한 텀들의 집합을 의미한다.

TRS의 룰을 정의할 때, 룰의 맨 왼쪽에 위치한 함수 기호를 *주 함수 기호(principal function symbols)* (혹은 *오퍼레이터*)이라고 부르고, 그렇지 않는 함수 기호는 *구성자(constructors)*라고 부른다. 룰의 왼쪽에서 주 함수 기호가 다른 함수의 인수에 위치하지 않도록 (즉 오직 구성자 만이 인수에 위치할 때) 정의되는 TRS를 *구성자 시스템(constructor systems)*이라고 한다.

함수형 프로그래밍에서는 적용(Applicative) TRS (혹은, Curried 함수)의 표현을 사용한다. 일반적인 TRS는 다음의 함수 *cur*를 이용하여 Curried 형태로 변환시킬 수 있다.

- $cur(x) = x$
- $cur(F(x_1, \ldots t_n)) = Ap(Ap(\ldots Ap(F, cur(t_1)), \ldots), cur(t_n))$

예를 들어, $cur(plus(x, y)) = Ap(Ap(plus, x), y)$ 가 된다. $Ap(M, N)$ 대신 infix 형식의 오퍼레이터 · 로 바꾸어 쓰면 $M \cdot N$로 표현되고, 여기서 다시 · 를 생략한 다음 괄호를 왼쪽부터 적용하기로 약속하면 함수형 언어에서 일반적으로 사용하는 Curried 함수의 표현을 얻는다. 예를 들어, $Ap(Ap(plus, x), y)$는 $plus\ x\ y$의 간단한 형태로 표현될 수 있다. 대부분의 함수형 언어는 ATRS의 구성자 시스템의 구조를 갖는다.

ATRS(Applicative TRS)의 주소는 *cur*이 적용되기 전의 TRS의 주소와 동일하다. 즉, 텀 $t \equiv F\ t_1, \ldots, t_n$와 주소 $u = i \cdot v$가 주어졌을 때 t의 부분 텀 $t|u$는 $t_i|v$가 되며 <>는 빈 주소를 의미한다.

문맥(context) C[ ]은 한 텀의 부분 텀을 □로 대치한 것을 의미한다. 예를 들어, $C[\ ] \equiv F\ \square\ A\ B$인 경우 $C[A] \equiv F\ A\ A\ B$가 된다. 일반적으로 여러 개의 □를 이용하는 문맥에서는 각 □ 마다 번호를 붙여 *복수 인수의 문맥(multiple ordered contexts)*을 구성할 수 있다. $C[\ ,\ ,\ ] \equiv F\ \square_1\ \square_2\ \square_3$ 인 경우, $C[A, B, C] \equiv F\ A\ B\ C$가 된다.

2.2 강 순차성과 Call-by-need 계산법

함수의 인수를 전송(parameter passing) 하는 일반적인 방법으로서 call-by-value와 call-by-name 방법이 있고, 이 중에서 call-by-name이 지연 계산법(lazy evaluation)을 위한 매개 변수 전송 방법임은 잘 알려진 사실이다. 그러나, 이 방법은 정의되는 함수의 인수(즉, formal parameters)가 모두 변수일 경우일 때만 적용될 수 있으며, Haskell과 같이 패턴(pattern)을 갖는 함수에는 적용할 수 없다. 이와 연관되어 제시되는 것이 call-by-need 계산법이다. Call-by-need는 TRS의 지연 계산법을 위해 제시된 방법으로서, 패턴을 갖는 함수형 프로그래밍 를 또한 TRS의 한 부류이므로 Haskell과 같은 함수형 프로그래밍에서도 이 방법을 적용할 수 있다.

Call-by-need 계산법에 대한 이론은 [6]에 잘 설명되어 있다. 이 이론은 매우 기술적이고 이론적이므로 여기에서는 그 주요 내용에 대해서 간단히 설명하고자 한다. 텀 t의 여러 레덱스들 중에서, 어떤 한 레덱스 d (혹은 d의 *잔여(residuals)*)를 축약하기 않고는 t의 정규형에 도달하지 못할 경우 d는 t의 *요구 레덱스(needed redex)*라고 부른다. 예를 들어, 룰 $F(x, A) \rightarrow 1$이 주어졌을 때, 한 텀 $F(R_1, R_2)$가 두 개의 레덱스 R_1, R_2를 내포하고 있는 경우 F-룰과의 패턴 매칭을 위해 R_2는 반드시 먼저 축약되어 그 결과가 A가 되는지를 확인해야 하지만, R_1는 그럴 필요가 없다. 따라서 R_2는 요구되지만, R_1는 그렇지 않다. 만약 룰이 $F(x, A) \rightarrow x$로 정의된다면 R_1의 잔여도 반드시 축약되어야 하므로 R_1 또한 요구 레덱스이다.

[6]에서 모든 OTRS (orthogonal TRS)의 정규형을 갖는 모든 텀은 최소한 하나 이상의 요구 레덱스를 갖는 다는 것이 증명되었다. 그러나 어떤 레덱스가 요구 레덱스인지는 모든 계산을 마치고 그 과정을 뒤돌아보았을 때만 알 수 있으므로, 일반적으로 OTRS에서 레덱스의 요구성을 판단하는 것은 불가능하다. 레덱스의 요구성(neededness)을 판단하기 위해서는 룰의 왼쪽과 오른쪽을 모두 고려해야 하지만 이것은 알고리즘의 복잡도가 너무 높아서 현실적으로 이용하기 어렵다. 따라서 룰의 왼쪽(left-hand side)만을 고려하여 요구성을 판단하는 방법이 제시되었는데, 이런 특성을 갖는 시스템을 *강 순차(strong sequential) 시스템*이라고 한다.

정의 3. (강 순차시스템) 정규형을 갖지만 아직 정규형이 아닌 텀 t에 대해서, t의 요구 레덱스가 존재하며 그 결정을 룰의 왼쪽만을 보고 결정할 수 있도록 정의되는 시스템은 강 순차 시스템이다.

강 순차 시스템은 OTRS의 한 부류이지만, 그 역은 성립하지 않는다. 예를 들어, 잘 알려진 Berry가 제안하는 다음과 같은 F-룰 $F(A, B, x) \to 1$, $F(B, x, A) \to 2$, $F(x, A, B) \to 3$ 은 OTRS이지만 강 순차 시스템이 아니다. 세 개의 레덱스를 갖는 텀 $F(R_1, R_2, R_3)$에서, R_1, R_2, R_3의 어떤 것도 요구 레덱스가 아니기 때문이다. 왜냐 하면, 첫 번째 룰에 대해서 R_1이 A이고 R_2가 B일 때 R_3는 계산하지 않아도 되며, 두 번째 룰과 세 번째 룰을 고려하면 같은 이유로 R_1과 R_2를 축약하지 않고도 정규형에 도달할 수 있으므로, 결과적으로 세 개의 레덱스 중에서 어떤 것도 요구 레덱스가 아니다.

어떤 한 텀의 요구 레덱스가 위치한 곳의 주소를 *지수(index)*라고 한다. 어떤 한 레덱스가 요구 레덱스인지를 판단하는 것은 그 레덱스가 위치한 곳의 주소가 지수인지를 판단하는 것이므로 지수를 찾는 것이 중요한 일이다.

정의 4. (Ω-리덕션과 지수) 새로운 두 개의 함수 기호 $\mathcal{G}$와 $\bullet$을 추가한다. $\mathcal{G}$는 아직 정의되는 않음을 의미하며, $\bullet$은 기존에 사용되지 않은 기호(fresh symbol)을 의미한다.

(1) 한 텀 t의 Ω-실례화는 t의 $\mathcal{G}$를 임의의 텀으로 치환한 것이다.

(2) 텀 t에서 $t|u$는 $\mathcal{G}$가 아니며 $t|u$의 Ω-실례화가 (일반 룰에 대한) 레덱스인 경우 다음과 같은 Ω-축약을 정의한다.

$$t \to_{\mathcal{G}} t', \text{ 여기서 } t' = t[u := \mathcal{G}].$$

(3) $\mathcal{G}$를 포함하는 정규형 t의 지수는 t의 한 주소 u로서 다음 조건을 만족한다.

$$t|u = \mathcal{G} \wedge t[u := \bullet] \not\to_{\mathcal{G}} \Omega.$$

2.3 분리가능성

분리가능성은 [8]에 자세히 논의되어 있다. 함수형 언어는 구성자 시스템의 구조를 가지므로, 본 연구에서는 일반적인 분리가능 시스템 대신 구성자 형태의 분리 가능성 시스템에 한정하여 논의한다. 이 경우 다루기가 훨씬 쉽고 강 순차성과의 연관성도 깊다.

정의 5. 주어진 텀들의 집합 $\mathcal{T}$의 모든 원소 t의 주소 u에 대해서 $t|u$가 진 부분 텀이고

각 $t|u$의 기호가 모두 동일하지 않으면, u는 $\mathcal{T}$의 *유용한 (useful)* 주소이다.

정의 6. 동일한 주 함수 기호를 갖는 룰들의 왼쪽 텀들로 구성된 집합 $l\mathcal{R}$를 고려한다.

(1) $l\mathcal{R}$에 대한 *분리 나무(Separation Tree)* U는 노드가 $l\mathcal{R}$의 텀들의 주소로 구성된 나무로서 다음과 같이 재귀적으로 정의된다.

- U의 뿌리 u_0는 $l\mathcal{R}$의 유용한 주소이고,
- u_0의 부분 나무들(subtrees)은 u_0에 위치한 기호들의 동일성을 기반으로 이들을 n개의 집합으로 분리하였을 때 $(l\mathcal{R} = l\mathcal{R}_1 \cup \ldots \cup l\mathcal{R}_n)$ 각 $l\mathcal{R}_i$의 분리 나무들이다.
- 분리 나무의 각 노드는 그의 조상에 같은 노드가 나타날 수 없도록 구성된다. 즉, 분리과정에서 한 주소는 오직 한 번까지만 이용될 수 있다.
- 위의 분리과정을 반복하여 $l\mathcal{R}$를 한 원소집합(singleton)으로 분리하였을 때의 분리 나무는 '*완전하다*'라고 한다.

(2) 어떤 한 구성자 시스템은 모든 주 함수 기호의 룰에 대해서 완전한 분리 나무를 가질 때 *분리 가능(separable)*하다고 한다.

예 7. 다음과 같은 구성자 시스템을 고려하자.

(i) Rules-1

$F\ x\ A\ B\ C \to 1$

$F\ B\ x\ A\ C \to 2$

$F\ A\ B\ x\ C \to 3$

(ii) Rules-2

$F\ x\ A\ B\ C \to 1$

$F\ B\ x\ A\ C \to 2$

$F\ A\ B\ x\ D \to 3$

(iii) Rules-3

$F\ x\ A\ B\ C\ (S\ D) \to 1$

$F\ B\ x\ A\ C\ (S\ D) \to 2$

$F\ A\ B\ x\ C\ (S\ E) \to 3$

(i)은 유용한 주소를 갖지 못하므로 분리가능하지 않다. (ii)는 유용한 주소 *4*를 기준으로, C를 포함하는 그룹과 D를 포함하는 그룹으로 분리되고, C를 포함하는 그룹은 다시 유용한 주소 *3*을 기준으로 다시 분리될 수 있으므로 이 룰은 분리 가능하다. (iii) 또한 분리 가능하며 그림 1과 같은 분리 나무를 구성할 수 있다

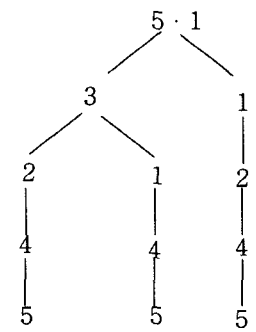

그림 1 분리 나무의 예

정리 8. [8] 한 구성자 시스템 $\mathcal{R}$이 강 순차성을 갖는다. $\Leftrightarrow$ $\mathcal{R}$은 분리성을 갖는다.

3. 함수형 프로그램의 TRS로의 변환

이 절에서는 함수형 프로그래밍 언어를 순수 선언적인 TRS로 변환하는 방법과 변환된 TRS의 특징에 대해서 논의한다. 변환을 적용하는 함수형 프로그래밍 언어를 문법적으로 구체적으로 정의하기보다는 함수형 프로그램 언어의 일반적인 특징을 개략적으로 제시하기로 한다.

3.1 변환되는 함수형 프로그래밍 언어의 조건

본 연구에서 논의되는 함수형 프로그램 언어 F의 특징에 대해서 소개한다. 여기서 F는 어떤 특정 언어라기보다는 언어의 스킴(scheme)으로서 다음과 같은 TRS의 구문적 특징을 갖는다. (함수형 언어 Haskell도 이 범주에 속한다.)

- 구성자 시스템 - F는 구성자 시스템(constructor systems)이다. 따라서 룰의 왼쪽에서 오퍼레이터는 맨 왼쪽 (즉, 주소 <>)에 단 한번 나타나며 나머지는 모두 구성자이거나 변수들로 구성된다.
- 좌 선형 - F 룰의 왼쪽 텀은 선형적(linear)이어야 하므로, 룰의 왼쪽에 같은 변수가 두 개 이상 나타날 수 없다.
- 적용시스템 - 함수의 표현은 적용시스템(applicative systems, 혹은 Curried functions라고도 함) 형태로 표현된다.
- 패턴 매칭 순서 - F의 패턴 매칭은 '위에서 아래로, 왼쪽에서 오른쪽' 방향의 순서대로 이루어진다.
- 지연 계산법(lazy evaluation)을 적용한다.
- 의미 없는 텀 - 예를 들어, fac True와 같이 정규형 t'이 오퍼레이터 기호(principal function symbols)를 포함하고 있는 경우, t' 의 값은 $\perp$로 표현하며 "의미 있는 값이 없다" (간략히 말해 "값이 없다")라고 한다. t'이 값이 없는 경우 $t \rightarrow^* t'$이 되는 모든 t 또한 값을 갖지 못한다. t가 값을 갖는 경우, t의 정규형이 t의 값이다.
- 군더더기 없는 룰 - 아래 예에서 처럼, 위에 위치한 룰의 왼쪽 텀 t_1이 아래있는 한 룰의 왼쪽 텀 t_2에 대해서 $\sigma(t_1) = t_2$, (즉, 실례화) 하는 경우는 고려치 않는다.

```
g x 1 = 1
g 8 1 = 2
```

Haskell의 경우 이것이 구문 에러는 아니지만 두 번째 룰은 아무런 역할을 하지 못한다. 또한 같은 패턴의 함수를 두 번 이상 정의되지 않는다.

- 사칙연산을 우선적으로 - 한 텀 안에 일반 레덱스와 사칙 연산에 의한 레덱스가 존재할 때 후자를 먼저 축약한다. 예를 들어, `2 * fac(2-1)`에서 일반 레덱스 `fac(2-1)`보다 사칙 연산 레덱스 `(2-1)`을 먼저 축약시킨다. 패턴 매칭의 순서와 함께 사칙 연산을 우선적으로 적용하면, `fac` 함수에서 `fac 1 = 0`와 같은 축약이 발생하지 않는다. ([1]에 의해서 논의됨).

3.2 함수형 프로그래밍에서의 순차성

일반적으로 람다 계산법으로 표현될 수 있는 함수는 순차적이어야(sequential) 한다는 것은 이론적으로 잘 설명되어 있다[5,7]. 한 함수가 순차적인지의 여부는 함수를 룰로서 표현할 때, 룰의 왼쪽 부분의 구조 (즉, 패턴) 의해서 결정된다. 순차적 조건을 만족시키지 못하는 대표적인 예로서 *Por* (parallel-or) 함수 *Por x T = T, Por T x = T, Por F F = F*와, Berry가 제시하는 *F* 함수 *F x A B = 1, F B x A = 2, F A B x = 3* 가 있다. 이런 함수들은 람다 계산법으로 정의될 수 없다. TRS의 관점에서 볼 때, *Por* 함수는 모호하므로 OTRS가 아니지만, *F* 함수는 OTRS임에도 불구하고 람다 계산법으로 표현될 수 없다.

예 9. (Haskell 프로그램)

```
data Cons       = A | B
f x A B         = 1
f B x A         = 2
f A B x         = 3
por x True      = True
por True x      = True
por x y         = False
loop            = loop
fac 0           = 1
fac x           = x * fac(x-1)
```

위에 정의된 예 9의 `f` 룰의 모습은 Berry의 F함수와 같다. (`por` 또한 parallel-or함수와 똑 같은 모습으로 정의될 수 있으나, 여기서는 변환을 위해 룰의 모습을 약간 수정하였다.) 그렇다면 "Haskell은 람다 계산법의 원리를 따르지 않는가?"라는 의문을 가질 수 있다. Haskell은 철저히 람다 계산법의 원리를 따르고 있다. Haskell 프로그램으로 정의된 룰의 외형적인 모습이 Berry의 F함수와 parallel-or와 같다고 하더라도, Haskell의 패턴매칭 전략때문에 이 함수들의 의미는 Berry의 F나 parallel-or와 다르다. 예를 들어, 어떤 한 의미없는 텀 $\perp$이 입력되더라도 *Por T* $\perp$는 *True*의 값을 갖지만, 이와 같은 형태의 Haskell의 텀 `por True loop`은 값을 갖지 못한다.

TRS 지연 계산의 기술적 논의에서 지수 (index)는 중요한 역할을 한다. F 함수가 순차적이지 못한 이유는, 예를 들어 Ω-텀 $F(\Omega, \Omega, \Omega)$에 대해서 Ω가 위치한 주소 *1, 2, 3* 중에 어떤 것도 지수가 되지 못하기 때문이다. 어떤 한 주소 u가 지수가 되려면 그와 연관된 모든

텀 t에 대해서 u가 유용한 주소가 되어야 한다. F함수는 주소 *1, 2, 3* 모두에 대해서 이 조건을 만족시키지 못한다.

TRS에서 어떤 한 텀 t가 지수의 위치에 종료되지 않는 레덱스 t'을 내포하고 있을 때, t'은 요구 레덱스이므로 t의 축약은 t'때문에 더 이상 진행되지 못하여 종료될 수 없는 현상이 발생한다. F의 '위에서 아래로, 왼쪽에서 오른쪽 방향으로'의 패턴매칭 특성때문에 TRS에서의 변수 역할과 지수의 원리가 그대로 적용되지는 않지만, F에서도 이와 같이 요구 레덱스에 의한 지수의 개념을 적용해 볼 수 있다.

위 함수에서 `loop`는 종료되지 않는 텀을 의미한다. 이것을 포함하는 텀 `f loop A B = 1, f A B loop = 3` 이지만 `f B loop A`는 종료되지 않는다. 또한 `por loop True = True`이지만 `por True loop`는 종료되지 않는다. 따라서 `f` 함수의 주소 *2*와 `por` 함수 주소 *2*는 지수라고 볼 수 있다. 만약 `por` 함수의 순서를 바꾼다면 `por` 함수의 지수는 첫 번째 인수가 된다. 또한 `fac` 함수의 주소 1은 지수이다.

`fac`의 *1*, `por`의 *2*, `f`의 *2* 모두 어떤 한 룰에 변수가 위치함을 볼 수 있으므로, 지연 계산적 관점에서 볼 때, 이런 위치에 있는 변수들은 TRS에서의 변수와 그 역할이 다름을 알 수 있다. 그러나 F에서의 변수의 역할이 언제나 TRS와 다른 것은 아니다. `f`의 *1*과 *3*이 지수가 될 없는 이유는 각각 첫 번째 룰과 세 번째 룰에 있는 변수가 TRS에서의 변수와 같은 역할을 하기 때문이다.

위에서 언급한대로 `f`의 두 번째 룰의 왼쪽 텀에 있는 변수 `x`와 `Por`의 두 번째 룰의 왼쪽 텀에 있는 변수 `x`는 TRS의 변수와 지연 계산의 동작의미가 다르므로, 이 룰을 TRS의 특성을 만족하도록 변환할 경우 이 변수들은 수정되어야 한다. 이러한 변수를 *유사 변수(semi-variables)*라고 부르기로 한다. 지연 계산 외에 고려해야 할 또 하나의 특성은 TRS의 직교성이다. 룰의 모호함에도 불구하고 패턴 매칭 전략에 따라 F는 직교성의 의미를 만족하고 있으므로 변환 과정에서 그 의미가 유지되도록 수정되어야 한다.

3.3 변환 알고리즘

이 절에서는 F를 TRS로 변환하는 함수 τ에 대해서 논의한다. TRS의 동작은 순수한 선언적 의미를 갖지만, F는 그렇지 못하다. F를 순수한 선언적 특성을 갖는 TRS로 변환하는 할 때, $\tau(F)$는 직교성을 가져야 하며, F의 요구성(neededness)이 $\tau(F)$에서도 그대로 유지되어야 한다. 이를 위한 변환 τ의 기본 개념은 주어진 룰 $\mathcal{R}$의 유사 변수와 그것을 포함하고 있는 룰을 찾아내어, 이 룰의 실례화 치환(instantiation substitution)을 통해 얻어지는 새로운 룰들을 $\mathcal{R}$에 추가하여 확장시키는 것이다(새로 생성되는 룰들의 수는 무한히 많을 수 있다).

함수형 프로그래밍의 '위에서 아래로, 좌에서 우 방향으로'의 패턴 매칭 순서를 고려하여 텀의 주소들 사이에서의 순서(ordering)를 다음과 같이 정의한다.

정의 10. (주소들의 순서)

두 주소 $u = i{\cdot}u'$와 $v = j{\cdot}v'$ 이 주어졌을 때 이 둘의 순서 $>_a$ 는 다음과 같이 정의된다.

(i) $u >_a v$ if $v = \langle\rangle$

(ii) $u >_a v$ if $i >_N j$ (자연수들 사이의 순서)

(iii) $u >_a v$ if $i = j \wedge u' >_a v'$

예를 들어, $4 >_a 2{\cdot}3$이며, $1{\cdot}2 >_a 1{\cdot}1$이 성립한다.

정의 11. 어떤 한 텀 t와 주소 u가 주어졌을 때, u보다 큰 주소의 위치에 있는 부분 텀들을 변수로 바꾸는 함수는 다음과 같이 정의된다.

(1) $V(u) = \{v \mid v \in O(t) \wedge (\forall v' >_a u \Rightarrow v \geq_a v')\}$
(u보다 큰 주소들 중에서 가장 작은 주소의 모음)

(2) $take(u, t) = t[v\_i := z_i]\ \forall v_i \in V(u)$. ($t|v_i$ 를 새로운 변수 z_i로 대치함)

예를 들어, $t \equiv K\ (C\ (C\ A\ B)\ D)\ (C\ (C\ E\ F)\ G)$ 이고 $u = 1{\cdot}1{\cdot}2$ 일 때, $V(u) = \{1{\cdot}2,\ 2\}$이므로, 새로운 변수 z_1, z_2를 가정하면 $take(u, t) = K\ (C\ (C\ A\ B)\ z_1)\ z_2$로 바뀐다.

τ 변환은 두 단계로 진행된다. 먼저 F의 룰에 있는 변수 중에서 유사변수를 찾아낸 다음, 이 변수를 포함하는 룰들을 실례화 확장(expansion by substitution) 시킨다.

알고리즘 12. (F에서 유사 변수 찾아내기) F로 정의된 한 함수에 대한 룰들의 집합 $\mathcal{R}$에 대해서, 프로그램으로 작성된 룰의 순서가 존중되는 왼쪽 텀들의 집합 $l\mathcal{R}$을 고려한다. $l\mathcal{R}$에 대한 좌표 $u_r \equiv (u, r)$는 $l\mathcal{R}$의 r번째 텀의 주소 u를 의미한다. $l\mathcal{R}$의 좌표 u_r에 위치한 어떤 한 변수가 유사 변수인지는 다음과 같이 결정할 수 있다.

(1) $UpperLeft(l\mathcal{R}, u_r) = \{take(u, t_i) \mid \forall t_{i \leq r} \in l\mathcal{R}\}$
(u_r을 기준으로, 그의 왼쪽-위쪽에 위치한 함수 기호만을 취함)

(2) $Core(l\mathcal{R}, u_r) = \{t \mid \forall t \in UpperLeft(l\mathcal{R}, u_r),\ t|u$는 변수 아님$\}$ ($UpperLeft$로 선택된 텀들의 각각에서 u에 변수가 위치한 텀들은 제외시킴)

(3) $Core(l\mathcal{R}, u_r)$의 r번째 텀을 t_r이라 하자.
$\exists t' \in Core(l\mathcal{R}, u_r)$ s.t. $t' \neq t_r \wedge t' \uparrow t_r \Leftrightarrow u_r$은 유사변수이다.
($t|u$는 진 부분 텀이므로 TRS의 요구성을 고려할 때 u_r는 변수가 될 수 없다.)

알고리즘 12를 적용하여 예 9의 룰들의 변수들이 유사변수인지를 점검해 보자. 먼저 `por` 룰의 세 번째 룰의 변수 `y` (즉, 2_3) 대해서, $UpperLeft(l\mathcal{R}, 2_3) = \{Por\ x\ True,\ Por\ True\ x,\ Por\ x\ y\}$가 된다. 여기서 두 번째

팀은 주소 *2*에 변수가 위치하므로 제거되며, 결과적으로 $Core(l\mathcal{R}, 2_3) = \{Por\ x\ True,\ Por\ x\ y\}$이 된다. 이 두 팀들은 모호하므로 2_3은 유사변수이다. 같은 원리로 por의 1_3, f의 2_2, fac의 1_2 모두 유사변수이다. 그러나 f의 1_1, 3_3, por의 1_1은 유사변수가 아니다.

알고리즘 13. ($\tau(F)$: 유사변수를 확장하여 TRS로 변환하기)

(1) 알고리즘 12을 적용하여 한 함수의 룰 $\mathcal{R}$에 대한 유사변수들을 구한다.
$U_R = \{u_r \mid t \in l\mathcal{R} \wedge t|u_r$ 은 유사변수$\}$

(2) 한 룰에 포함된 여러 변수들을 동시에 치환하는 함수 σ를 가정한다. $R_i \in R$에 있는 유사변수들 모두를 모호성을 발생시키지 않는 범위에서 치환 확장(expansion by substitution)시킨다. $Exp(R_i) = \{\sigma(R_i) \mid \forall R' \in R,\ R'$ not $\uparrow\ \sigma(R_i)\}$ 여기서 치환함수 σ의 치역(range)은 정의된 함수의 타입을 만족시키는 구성자로서 무한히 많을 수 있다(예를 들어, fac 함수의 경우).

(3) 유사변수를 포함하는 각 $R_i \in R$를 $Exp(R_i)$로 대치시킨다. 이 과정에서 중복 룰들은 그 중에 하나만 취한다.

(4) F에 정의된 모든 함수의 각 룰에 대해서 위의 과정을 적용한다.

Factorial 함수 fac에 대해서 τ 변환을 적용한 결과 $\{Fac\ 0 = 1,\ Fac\ n{:}[INT\text{-}0] = n * Fac(n-1)\}$를 얻는다. fac 함수의 1_2의 변수 n은 유사변수이므로 치환 확장된다. 여기서 $n{:}[INT\text{-}0]$ 표현은 $Fac\ n$의 룰에서 0을 제외한 모든 정수에 대해서 치환 확대되는 무한 개의 룰을 구조적으로 표현한 것이다. 이 룰은 직교적이며 강 순차성을 갖는다.

f 함수의 룰에서, 두 번째 룰에 있는 변수 x는 유사변수이므로 다음과 같이 확장된다.

$F\ x\ A\ B = 1$
$F\ B\ A\ A = 2$
$F\ B\ B\ A = 2$
$F\ A\ B\ x = 3$

변환된 TRS는 직교적이며, 강순차적인 특성을 갖는다. 예를 들어, 룰 F에 대한 $\mathcal{L}$ 팀들은 다음과 같이 지수(밑줄로 표시)를 갖는다. - $F(\mathcal{L}, \underline{\mathcal{L}}, \mathcal{L})$, $F(B, \underline{\mathcal{L}}, \mathcal{L})$, $F(A, \underline{\mathcal{L}}, \mathcal{L})$, $F(\mathcal{L}, A, \underline{\mathcal{L}})$, $F(\underline{\mathcal{L}}, B, \mathcal{L})$, $F(\mathcal{L}, \underline{\mathcal{L}}, B)$, $\underline{F(\mathcal{L}, \mathcal{L}, A)}$.

por 함수의 룰에서, 첫 번째 룰에 있는 변수 x를 제외한 모든 지수는 유사변수이다.

$Por\ x\ True = True$
$Por\ True\ False = True$
$Por\ False\ False = False$

이 por 함수에 대해 τ 변환을 적용하는 중간 과정에서 중복되는 룰들이 발생하고 있으며, 이 룰들은 제거된다.

예 2의 g 함수에 대한 룰에 있어서, 세 번째 룰에 있는 변수 z는 유사변수이므로 다음과 같이 확장된다.

$G\ Nil\ Nil = 1$
$G\ (Cons\ x\ y)\ Nil = 2$
$G\ Nil\ (Cons\ x\ y) = 3$
$G\ (Cons\ a\ b)\ (Cons\ x\ y) = 3$

예 2의 h는 TRS로 변환되더라도 그 룰을 그대로 유지한다.

3.4 변환된 TRS의 특성

도움정리 14. 어떤 한 함수에 대한 룰 $\mathcal{R}$과 그의 변환 $\tau(\mathcal{R})$을 고려하자.

(i) $(t \in \tau(l\mathcal{R})) \mid u$가 변수이면, $(t \in l\mathcal{R}) \mid u$ 또한 변수이다.

(ii) $(t \in l\mathcal{R}) \mid u$가 유사변수이면 $\tau(\mathcal{R})$에서 t의 u는 지수이다.

증명. (i) 변환 τ의 정의를 고려할 때 당연히 성립한다.

(ii) t의 u가 지수가 아니라고 가정하자. 그러면, $t[u := \mathcal{L}]$는 $\tau(l\mathcal{R})$에 대해서 Ω-레덱스이다. u가 지수가 아니므로 $t[u := \bullet]$ 또한 Ω-레덱스가 되어야 한다. 그렇다면, $(t' \in \tau(l\mathcal{R})) \uparrow t$인 t'이 존재해야 하나, 알고리즘 12.3에 따라 이것은 성립할 수 없다. 따라서 $\tau(\mathcal{R})$에서 t의 u는 지수이다. □

변환 알고리즘 τ는 함수형 언어 F의 축약 관계와 요구성을 유지시키고 있다.

도움정리 15. F의 조건을 만족하는 프로그램 P를 $\tau(P)$로 변환하였을 때 다음이 성립한다.

(i) 팀 $t, t' \in P$에 대해서 $t \rightarrow_P^* t'$이고 이들이 값을 가지면, $t \rightarrow_{\tau(P)}^* t'$이다.

(ii) 한 Ω 팀 t의 주소 u에 대해서 다음이 성립한다.
t의 u가 P에서 지수이다. $\Leftrightarrow$ t의 u가 $\tau(P)$에서 지수이다.

위의 도움정리 (i)에서 그 역은 성립하지 않을 수 있다. 예를 들어, $\tau(\mathcal{R})$에서 t'이 주함수 기호를 포함하는 경우, τ(P)의 t'와 P의 t'는 다르다.

τ 변환에 있어서 유사변수들의 선택 및 그들을 치환 확장하는 데 있어서의 특징에 대해서 논의해 보자. F의 조건을 만족하는 룰 $\mathcal{R}$에서 어떤 한 변수에 대한 유사변수 여부를 결정하는 것은 $l\mathcal{R}$에서 그 변수가 위치한 곳의 위쪽-왼쪽(upper-left)의 패턴만을 고려하였을 때, 모호성이 발생하는 가에 의해서 결정된다. 모호성이 발생하게 되면 그 변수는 유사변수로 판단하고, 유사변수를 치환 확장함으로써 $\tau(\mathcal{R})$에서는 모호성이 발생되지 않도록 한다.

$l\mathcal{R}$에서 유사변수가 발생하는 위치는 다음과 같은 특징을 논의하기 위해 다음과 같은 예를 고려해 보자.

예 16. (유사변수를 결정하는 F룰의 구조)

```
p A x B 5 ··· = ···   q x 2   = ···   r 1 ··· = ···
p A 1 B 6 ··· = ···   q 1 x   = ···   r 2 ··· = ···
p A 2 B 7 ··· = ···   q ··· ··· = ···   r x ··· = ···
p A 3 B x ··· = ···
```

예 16에서 A와 B는 동일한 기호를 의미하기 위해서, 숫자는 다른 기호를 표현하기 위한 용도로 사용되고 있다. 필요에 따라 동일한 기호를 포함하는 열은 생략될 수 있다. p 마지막 룰의 x는 첫 번째 룰과 단일화(unification)될 수 있으므로 유사변수가 된다. q룰의 구조는 p 룰의 특수한 경우로서, p에서 같은 기호로 된 열이 생략되었고 유사변수에 영향을 주지 않는 룰들을 고려하지 않는 형태이다. 따라서 q 두 번째 룰의 변수 또한 유사변수이다. r 룰은 p의 4_4에 있는 변수의 유사변수 여부를 결정하는 데 있어서, 동일한 기호를 갖는 첫 번째 열을 제외시키고, 변수를 갖는 두 번째 열 또한 모든 형태의 기호로 실례화(instantiate)가 될 수 있으므로 제외시킨 경우에 해당된다.

예 9에서 f룰의 2_2, por의 2_2, 2_3은 예 16의 q룰의 경우에 속하며, por의 1_3과 fac의 1_2는 위의 $\mathcal{R}$의 경우에 속한다. f의 3_3은 위의 경우에 속하지 않으므로 유사변사가 아니다.

이 현상을 복수 인수의 문맥을 이용하여 표현하면 그 모습이 더욱 명확해 진다. $C[\ \square_1, \square_2] \equiv p\ A\ \square_1\ B\ \square_2$ 라고 하자. 그러면 예 16의 r은 각각

```
C[x, 5]
C[1, 6]
C[2, 7]
C[3, x]
```

으로 표현될 수 있다. 위의 문맥 표현에서 변수가 포함된 첫 번째와 세 번째를 선택하고 유사변수가 τ 변환 과정에서 치환 확장되는 것을 고려하면, 예 16의 룰들을 τ 변환시켰을 때, 위와 같은 형태의 문맥은 존재하지 않게 된다. 일반적으로 다음과 같은 원리를 유추할 수 있다.

도움정리 17. 하나 이상의 룰들로 구성된 $RS \subseteq \tau(\mathcal{R})$의 왼쪽 부분을 복수 인수의 문맥 C[,···,]으로 표현하기로 하자. 이때 다음과 같은 형태의 문맥은 존재하지 않는다(여기서, R_i, ···, R_z는 RS의 오른쪽 부분이고, x는 변수이며, M들은 RS 왼쪽의 부분 텀을 의미한다).

$C[x, M_{i2}, M_{i3}, \cdots, M_{ik}] = R_i$
$C[M_{j1}, x, M_{j3}, \cdots, M_{jk}] = R_j$
$C[M_{l1}, M_{l2}, x, \cdots, M_{lk}] = R_l$
$\cdots \quad \cdots \quad \cdots \quad \cdots$
$C[M_{z1}, M_{z2}, M_{z3}, \cdots, x] = R_z$

도움정리 17로부터 다음과 같은 원리를 유추할 수 있다.

도움정리 18. $RS \subseteq \tau(\mathcal{R})$의 왼쪽 부분을 복수 인수의 문맥으로 표현할 때, 모든 문맥은 최소한 하나의 유용한 주소를 갖는다.

정리 19. (함수형 언어와 TRS의 연관성)

(1) 함수형 언어 F의 조건에 따라 정의된 한 함수의 룰 $\mathcal{R}$에 대해서 $\tau(\mathcal{R})$은 OTRS이다.
(2) $\tau(\mathcal{R})$는 분리성을 갖는다.
(3) $\tau(\mathcal{R})$은 강 순차성을 갖는다.

증명. (1) F의 군더더기 없는 룰의 조건, 알고리즘 12와 알고리즘 13에 따라 $\tau(\mathcal{R})$은 직교성을 갖는다.
(2) 도움정리 18에 따라, 맨 처음 $\tau(\mathcal{R})$의 왼쪽 부분에서 유용한 주소의 기호들을 기준으로 $\tau(\mathcal{R})$을 분리한 다음, 이 과정을 각 그룹의 룰이 하나가 될 때까지 반복한다.
(3) $\tau(\mathcal{R})$는 분리성을 가지며, 정리 8에 의해서 $\tau(\mathcal{R})$은 강 순차성을 갖는다. □

예 2에서 두 함수 h와 g의 인수의 형태가 서로 대칭적 구조를 이루고 있더라도 실제 동작 의미는 그렇지 않음으로써 예상했던 c g = f의 등식 추론이 성립되지 않음을 보았다. 그러나, 이 두 함수의 룰들을 τ로 변환시켰을 때, 이 둘의 인수들이 대칭적 구조를 이루지 않음을 알 수 있다. 따라서 등식 c g = f이 성립하지 않음은 당연하다. 함수형 프로그래밍 F의 어떤 한 프로그램 P가 등식 추론의 원리를 갖기 위해서는 TRS 지연 계산의 의미를 만족시켜야 한다; P = $\tau(P)$이면 P는 등식 추론의 원리를 만족시킨다.

3.5 람다 계산법으로의 변환

함수형 언어 F의 조건을 만족하면서 작성된 프로그램 P는 분리성을 갖는 TRS $\tau(P)$로 변환될 수 있고, 이들은 다시 람다 계산법으로 변환될 수 있다.

정리 20. (함수형 언어와 람다 계산법의 연관성)

F의 조건을 만족하는 프로그램 P와 텀 t, $t' \in P$에 대해서, $t \rightarrow_P t'$이고 이들이 값을 가지면, $\phi(t) \rightarrow_\beta^* \phi(t')$인 람다 계산법으로의 변환 ϕ가 존재한다.

증명. 도움정리 15에 따라 변환 τ는 함수형 프로그래밍의 축약 관계를 보전하고, 정리 19.(2)에 따라 P는 분리성을 가지며, [9]의 원리에 따라 모든 분리성을 갖는 TRS는 람다 계산법으로 변환될 수 있다.

정리 8에 따라 강순차성의 특징을 갖는 구성자 TRS는 분리성을 갖는다는 것이 증명되었다. 앞서 논의된 대로 함수형 프로그램 언어 F는 분리성을 갖는 TRS로 번역될 수 있고, 따라서 람다 계산법으로 번역될 수 있다. 분리성 TRS에서 람다 계산법으로의 변환과 관련된 이론은 [9]에 소개되어 있으므로 구체적인 논의는 생략하기로 하며, 여기에서는 예 9에 소개된 f 함수를 τ 변

환한 F를 예로 들어 설명하기로 한다.

$F\ x\ A\ B \quad = 1$

$F\ B\ A\ A \quad = 2$

$F\ B\ B\ A \quad = 2$

$F\ A\ B\ x \quad = 3$

위에 정의된 TRS에 대해서 $F\ B\ B\ A$를 계산하기 위해서는 먼저 이를 매칭하는 룰을 찾아야 한다. 그 룰을 찾는 과정은 F 룰들을 분리함으로써 이루어진다. 위의 F룰에서 주소 2가 유용한 주소로서, 룰 왼쪽에 있는 각 텀들은 각각 2에 위치한 기호 A와 B에 따라 두 그룹으로 분리된다 - $\{F\ x\ A\ B,\ F\ B\ A\ A\}$와 $\{F\ B\ B\ A,\ F\ A\ B\ x\}$. 이 둘 중에서 $(F\ B\ B\ A)\ |\ 2 = B$이므로 두 번째 그룹이 선택된다. 이 그룹에서 다시 반복적으로 주소 2를 제외한 유용한 주소를 찾게 되면 1이 선택되고, 이 그룹은 다시 1에 위치한 기호에 따라 다시 두 개의 그룹으로 분리된다 - $\{F\ B\ B\ A\}$와 $\{F\ A\ B\ x\}$. $(F\ B\ B\ A)\ |\ 1 = B$ 이므로, 첫 번째 그룹이 선택되는데, 이 그룹은 한 원소 집합 (singleton)이므로 더 이상의 분리 과정 없이 이에 대한 룰 (위에 정의된 룰 중에서 세 번째)을 선택할 수 있다. 이 과정을 람다 텀으로 코딩함으로써 위의 TRS F는 람다 계산법으로 번역될 수 있다.

위의 F룰처럼 분리성을 갖는 모든 TRS는 람다 계산법으로 코딩될 수 있다. [9]에서는 Böhm이 제시한 Böhm-out 변환 원리 [10]를 적용하여, 일반적인 모든 분리 과정이 두 함수 Permutation 함수 **P**와 Selection 함수 **U**로 코딩될 수 있음을 보여주고 있다.

4. 관련 연구 및 결론

'위에서 아래로, 왼쪽에서 오른쪽' 방향으로 패턴 매칭을 함으로써 발생할 수 있는 함수형 프로그래밍 언어 의미의 모호함과 TRS와의 연관성에 대한 연구는 Kennaway에 의해서 이루어진 적이 있다[1]. 그는 함수형 프로그래밍 언어가 구성자 시스템으로 한정되지 않고 좀 더 일반적인 형태의 패턴을 갖도록 정의할 수 있는 가능성을 제시하였다. 한편, 함수형 프로그래밍의 패턴 매칭은 *우선 순위를 갖는 축약(priority rewriting)*을 갖는 시스템의 개념과 연관될 수 있다[11,12].

본 연구에서는 패턴 매칭과 지연 계산법을 적용하는 구성자 시스템 구조의 함수형 언어를 TRS로 변환하여 그 특징을 파악하였다. 지연 함수형 언어는 분리성을 갖는 TRS로 변환될 수 있으며, 분리성의 원리에 따라 람다 계산법으로 번역될 수 있다. 이 변환은 요구성의 원리를 보전하며, 잘 정립된 강 순차성의 이론을 적용할 수 있다. 이 두 변환은 지연 계산 함수형 언어의 의미를 람다 계산법으로 설명하는 이론적 근거를 제공하고 있다. 다른 측면에서 볼 때, 본 연구는 함수형 프로그래밍의 등식 추론에서의 한계와 적용 범위를 명확히 파악하는 데 이용될 수 있다.

참 고 문 헌

[1] Richard Kennaway, The Specificity Rule for Lazy Pattern-Matching in Ambiguous Term Rewriting Systems. *ESOP '90*, Lecture Notes in Computer Science No. 432, Springer-Verlag, 1990.

[2] Simon L. Peyton Jones, *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[3] Simon Thompson, *The Craft of Functional Programming*, 2nd Edition. Addison-Wesley, 1999.

[4] J.W. Klop, Term rewriting systems. In Abramsky et al., editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

[5] H. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[6] G. Huet and J.-J. Lèvy, Computations in Orthogonal Rewrite Systems I and II. In Lassez and Plotkin, eds., *Computational Logic: Essay in Honor of Alan Robinson*, MIT Press 1991. (Originally appeared as Technical Report 359, INRIA, 1979.)

[7] G. Berry. Sèquentialitè de l'èvaluation formelle des λ-expressions, in *Proc. 3-e Colloque International sur la Programmation*, Paris, 1978.

[8] 변석우, 분리가능 시스템의 지수 추이성과 변환. *정보과학회 논문지: 소프트웨어 및 응용*, 제31권 제5호, 659-666쪽, 2004년 5월.

[9] S. Byun, J.R. Kennaway, and R. Sleep, Lambda-definable Term Rewriting Systems, *Lecture Notes in Computer Science* 1023, pp. 106-115, Springer-Verlag, 1996.

[10] C. Böhm, Alcune proprietà delle forme β-η-normali nel λ-K-calcolo. *IAC Pubbl.*, 696:19, 1968.

[11] A. Laville, Lazy pattern matchingin the ML programming. INRIA report 663, 1987.

[12] Y. Toyama, S. Smesters M. van Eekelen, and R. Plasmeijer, The Functional Strategy and Transitive Term Rewriting Systems. In (eds. Sleep, Plasmeijer, and van Eekelen) *Term Graph Rewriting*, John Wiley and Sons Ltd. 1993.

변 석 우

1976년~1980년 숭실대학교 전자계산(학사). 1980년~1982년 숭실대학교 전자계산(석사). 1982년~1999년 ETRI 책임연구원. 1988년~1994년 영국 University of East Anglia 전산학(박사). 1998년~현재 경성대학교 컴퓨터정보학부 교수. 관심분야는 rewrite system, 함수형 프로그래밍, 의미론, 정형 시스템 등