□ 기술해설 □

Rewriting 기반 프로그래밍

한국전자통신연구소 변석우*

● 목 차 ●

1. 서 론
2. Orthogonal Term Rewriting Systems
2.1 Term Rewriting Systems
2.2 Orthogonal TRS
2.3 전략(Strategies)
3. 람다 계산법(Lambda Calculus)
4. 연구 동향 및 결론

1. 서 론

'같음(equivalence)'의 개념은 수학 및 프로그래밍에서 가장 기초적인 개념으로서 reflexive, symmetric, transitive의 논리적 관계(relation)로 정의되고 있다. Rewriting은 reflexive, transitive 관계를 가지고 있는 데, 직관적으로 방향성(direction)을 갖는 같음의 관계로서 생각할 수 있다. Rewriting 관계를 → 로 나타냈을 때, $a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n$이 주어졌다고 하자. 직관적으로 이야기하여, a_n이 우리가 원하는 답이라고 할 때 a_2는 a_1보다 한 단계 답에 접근한 형태로 볼 수 있다. 이와 같이 rewriting은 계산이 수행됨에 따라 답에 접근한다는 단순한 계산논리를 기반으로 하고 있다 — 특히, 또 하나의 잘 알려진 계산 방법인 unification보다 간단 명료하다. Rewriting의 계산논리는 함수적 개념과 쉽게 연관되어 있다. 함수는 독립변수(입력)에 대하여 언제나 일정한 종속변수(출력)를 갖는 관계로서 정의된다. Rewriting에서 계산되어 얻어지는 답이 유일한 경우에 이 계산 기법은 함수적 의미를 적절히 표현하는 계산 기법으로서 받아들여 질 수 있을 것이다. Rewriting의 이와 같은 특성을 confluent(혹은 Church-Rosser)라고 하는데, 이것은 계산 결과에 대한 안전성(soundness)의 의미를 내포하고 있으므로 rewriting의 여러 특성 중 가장 중요하다고 보여진다. 수학에서 함수의 개념이 가장 중요한 기초적 개념으로 이용되고 있는 것과 마찬가지로, 컴퓨터 사이언스에서도 함수적 개념의 중요성은 절대적이다. 특히, 프로그래밍 언어 의미론에서 프로그램의 의미를 함수적 의미로서 표현하는 denotational 의미론이 가장 중요한 의미론으로서 받아들여지고 있다. 이런 측면에서 볼 때, rewriting은 함수 프로그래밍(functional programming)은 물론 명령형 프로그래밍(imperative programming)에 대한 동작 의미로서도 이용될 수 있다.

본 논문에서는 함수에 대한 rewriting 기법으로서 대표적으로 이용되고 있으며 Church-Rosser의 특성을 만족하는 람다 계산법 (lambda calculus)과 orthogonal TRS(term rewriting systems)의 기본 원리 및 응용 가능성에 대하여 논의하고자 한다. 1930년대에 Church에 의해서 창조된 람다 계산법은 함수에 대한 대표적 rewriting 시스템으로서 약 60년의 역사를 가지고 있다. 개발 당시에는 타입(type)이나 의미(model)를 고려하지 않은 채 구문이 정의되었으나, 1970년경에 Dana Scott에 의해서 의미가 주어지고 그 이론이 프로그래밍 언어 의미론에 적용되었으며, 최근에는 타입에

*정회원

대한 많은 연구가 이루어지고 있다. 람다 계산법에서 사용되는 기호는 변수와 λ뿐이며 룰 또한 β를 단 하나만으로 정의된다. 이와 같은 구문의 단순함은 이론적인 측면에서는 매우 좋은 장점이 되고 있다. 그러나 람다 계산법을 실용적으로 직접 사용하기에는 너무 추상화되었다고 보여진다. 람다 계산법은 함수 언어를 구현하기 위한 중간 언어(intermediate language)로서 사용되기도 하는 데, 이 경우 순수 람다 계산법이 아닌 자연수나 부울 대수(Boolean algebra)에 대한 기호도 함께 쓰는 확장된 람다 계산법의 형태로서 이용되고 있다 [1].

TRS에서는 원하는 만큼의 기호를 자유롭게 정의하여 사용하고 있으며, 따라서 많은 TRS가 정의되어 사용되고 있다. 람다 계산법, Combinatory Logic, 함수 프로그래밍 언어(functional programming languages) 등이 모두 이 범주에 속한다. 본 논문에서는 orthogonal TRS 및 순차성(sequentiality) 등에 대해서 논의한 다음, 람다 계산법에 대해서 논의한다. 본 논문은 이 분야에 경험이 없는 독자의 이해를 돕기 위해 서술 식으로 기술된다. 이 분야에는 좋은 자료들이 많이 있는 데, TRS에 대해서는 [2][3], 람다 계산법에 대해서는 [4][5]를 참조하기 바란단. 또한 이들의 효율적인 구현 방법과 관련되고 있는 그레프(graph) rewriting에 대해서는 [6][7]을 참조하기 바란다.

2. Orthogonal Term Rewriting Systems

2.1 Term Rewriting Systems

TRS는 기호(symbols), 텀(terms) 및 룰(rewrite rules)의 세 가지 요소로 구성된다. 일반적으로 기호는 변수(variables)와 함수기호 (function symbols)가 사용되는 데, 함수기호들은 인수의 수에 따라 구분될 수 있다. 함수 기호 중에서 인수의 수가 0인 것을 상수(constants)라고 부른다. 사용되는 기호가 정의되면 이것을 기반으로 텀을 정의할 수 있다. 텀은 귀납적(inductively)으로 정의된다. 모든 변수들은 텀이 될 수 있고, F가 n개의 인수를 갖는 함수 기호라고 할 때 텀 $t_1, ..., t_n$에 대하여 $F(t_1, ..., t_n)$ 또한 텀이다. 텀이 결정되면 룰을 정의할 수 있다. 룰은 두 개의 텀(t_1, t_2)의 의해서 정의되는 데, $t_1 \rightarrow t_2$ 라고 쓴다. 이때 왼쪽의 텀 t_1은 변수일 수 없으며, 오른 쪽의 텀 t_2에 사용되는 모든 변수는 t_1에 반드시 나타나야 한다. 텀 중에서 변수를 포함하지 않는 텀을 닫힌 텀(closed term, 혹은 ground term)이라고 한다. 또한 귀납적으로 정의되는 텀 $F(t_1, ..., t_n)$에서, $t_1, t_2, ... t_n$ 는 모두 $F(t_1, ..., t_n)$의 부분텀(subterm)이라고 부른다. 물론 모든 t_i의 부분텀 또한 $F(t_1, ..., t_n)$의 부분텀이다.

프로그래밍의 입장에서 볼 때, 정의된 룰은 프로그램이라고 생각할 수 있다. 계산하고자 하는 대상은 텀으로 표현되며(특히 대부분의 경우 닫힌 텀으로 표현됨) 텀은 정의된 룰에 따라서 rewriting 됨으로써 계산된다. 텀에서 변수는 '임의의 텀'으로 바뀔 수 있으며, 이러한 현상을 실례화(instantiation)이라고 한다. 특히 정의된 룰의 왼쪽 텀이 실례화된 텀을 레덱스(redex, reducible expression)이라고 부른다. 예를 들어, $H(x, 1) \rightarrow G(1, x)$이라고 룰이 정의되었다고 하자. 이 경우, 텀 $H(2, 1)$은 그 룰의 레덱스이다. 레덱스가 정의될 때 변수 x가 2로 바뀌었음(substitution)을 주목하라. 레덱스는 해당되는 룰의 오른쪽 텀으로 대칭 됨으로써 rewrite된다. 이때 오른쪽 텀 역시 앞서 정의된 바뀜이 적용된다. 따라서 $H(2, 1) \rightarrow G(1, 2)$가 된다. 이러한 현상은 다음과 같이 좀 더 정형화하여 표현될 수 있다. 바뀜을 변수로부터 텀으로의 사상 함수 σ: Var $\rightarrow$ Term로서 정의하자. 이 함수는 텀 내의 모든 변수에 대하여 응용되므로, $\sigma(F(t_1, ..., t_n)) = F(\sigma(t_1), ... \sigma(t_n))$이 성립한다. 위의 경우 $\sigma(x)=2$로 정의되어, $\sigma(H(x, 1)) = H(\sigma(x), 1) = H(2, 1)$이 된다. 이와 같이 어떤 한 룰 $t_1 \rightarrow t_2$이 주어졌을 때 대하여 그 룰의 레덱스는 한 바뀜 함수 σ에 의하여 $\sigma(t_1)$으로서 표현된다.

일반적으로 계산의 대상인 텀은 그 자체가 레덱스라기 보다는 레덱스를 부분텀으로 갖는 텀이다. 즉, $H(x, 1) \rightarrow G(1, x)$의 룰이 정의

되었을 때, 텀 S(H(2, 1))의 부분텀 H(2, 1)은 레덱스이다. 이 경우 레덱스를 제외한 텀의 나머지 부분을 문맥(context)으로서 정의할 수 있다. TRS에서 rewriting은 문맥의 구조를 유지한다. 즉. S(H(2, 1)) → S(G(1, 2)) 이 성립한다. 이 상황은 정형화하여 다음과 같이 표현된다. 문맥은 하나의 빈 공간 □을 포함하며 C[ ]로서 나타낸다. C[ ]에서 □이 어떤 텀 M으로 바뀐 것은 C[M]으로 표현되는 데, C[M]은 텀이 된다. 이때 M이 어떤 룰 $t_i \rightarrow t_j$의 레덱스라면 $C[\sigma(t_i)] \rightarrow C[\sigma(t_j)]$의 rewriting이 발생하게 된다. 예를 들어 위의 예에서 C[ ] = S(□)이 된다. 이러한 문맥의 개념은 당연해 보일 수 있으나, 매우 중요한 의미를 내포하고 있다. 앞서 언급한 대로 rewriting을 같음에 대한 계산법으로 생각할 때, 어떤 문맥에서도 이러한 같음의 개념이 성립될 수 있으며, 따라서 문맥에 포함된 한 레덱스는 같은 값을 나타내는 단순화된 레덱스로서 대치되어도 좋다는 개념이 성립되고 있다. 어떤 한 텀 t가 레덱스를 포함하고 있지 않으면 t는 정규형(normal form)이라고 불리며, 한 텀 s가 정규형 s′ 으로 rewrite될 수 있으면 s는 정규형을 갖는다고 말한다. 직관적으로 정규형 s′는 텀 s의 값(value)이라고 간주할 수 있다.

2.2 Orthogonal TRS

앞서 소개된 TRS는 좀 더 구체적으로 정의되어 이용된다. 이 TRS는 요구되는 특성에 따라서 제약된 형태의 룰을 갖게 된다. 중요한 개념들로서 "어떤 한 텀이 언제나 단 하나의 텀으로 rewrite되어 수렴(converging)될 수 있는가?(CR, Church-Rosser)", "모든 텀이 언제나 정규형으로 rewrite될 수 있는 가?(SN, strongly normalizing)", "한 텀이 정규형을 갖는 다면 그 정규형으로 rewrite될 수 있도록 레덱스들을 선택할 수 있는 전략(strategy)가 존재하는 가?(existence of normalizing reduction strategy)" 등의 특성이 존재한다. TRS는 이러한 요구되는 특성을 만족하도록 제한되어 정의된다. SN은 바람직한 특성이기는 하지만 경우에 따라서는 너무 강한 조건이라고 생각되며, 이 특성이 언제나 요구되지는 않을 것이다. 예를 들어, 운영 체제와 같이 계산이 종료되지 않고 계속 진행되어야만 하는 응용 분야는 SN이 될 수 없기 때문이다. 대부분의 함수 언어에서는 일반적으로 SN을 요구하지 않고 있다.

정규형 전략(*normalizing reduction strategy*)에 대해서는 다음 절에서 논의하기로 하고 본 절에서는 CR에 대하여 논의하고자 한다. CR의 특성을 만족하는 대표적인 TRS로서 orthogonal TRS가 존재한다. Orthogonal TRS는 두 가지 조건 *left linear*와 *non-overlapping*을 만족해야 한다. 첫번째 조건은 룰의 왼쪽 텀에 같은 변수가 두번 또는 그 이상 나타날 수 없는 경우이다. 두번째 조건은 룰의 왼쪽 텀의 어떤 두 부분텀이 같은 하나의 텀으로 실례화될 수 없음을 의미한다. 예를 들어 다음과 같은 룰을 생각해 보자 : If(T, x, y) → x, If(F, x, y) → y, If(x, y, y) → y. 이 룰은 두 조건 중에 어느 하나도 만족시키질 못하고 있다. 세번째 룰의 왼쪽에 같은 변수 y가 두번 나타났으므로 left-linear가 아니며, 첫번째 룰과 세번째 룰, 그리로 두번째 룰과 세번째 룰은 overlap되고 있다. 예를 들어. If(T, x, y)와 If(x, y, y)은 같은 텀 If(T, T, T)로서 실례화 될 수 있다. 따라서 텀 If(T, T, T)이 rewrite되려고 할 때 선택되는 룰은 첫번째 룰이 될 수도 있고 세번째 룰이 될 수도 있다. 또 하나의 overlapping 예로서 L(L(x)) → 0이 있다. 이 경우 왼쪽 텀의 두 부분 텀 L(L(x))와 L(x)는 같은 텀 L(L(0))으로 실례화될 수 있으며, 텀 L(L(0))을 rewrite 할 때 바깥쪽의 레덱스와 안쪽의 레덱스중에 어느 하나가 비결정적(non-deterministic)으로 선택될 수 있다. Non-overlapping의 조건은 이러한 non-deterministic이 일어나지 않도록 한다. 이 두 조건을 만족하는 모든 TRS는 CR이다. CR을 만족하면 계산 순서를 변경하더라도 결과 값이 언제나 일정하므로, CR의 특성을 갖는 언어를 이용하여 병렬 프로그래밍 하는 경우 계산순서의 제어(flow control)의 문제가 자동적으로 해결된다. 따라서 제어 병렬성(control parallelism)에 매우 유익하게 이용될 수 있다.

위에 정의한 If룰은 orthogonal은 아니지만 사실 CR의 특성을 갖는다. 즉, orthogonal이 CR의 모든 경우를 전부 포함하고 있지는 않다. 그러나 orthogonal이 갖는 장점은 룰의 왼쪽만을 고려하여 CR을 정의될 수 있다는 점이다. 일반적으로 한 TRS의 CR 여부를 점검하기 위해서는 룰의 왼쪽과 오른쪽 모두를 고려해야 하는 데, 이와 같이 룰의 왼쪽과 오른쪽을 모두 고려하여 CR의 여부를 파악하기는 매우 어렵다. TRS를 정의할 때는 이와 같이 계산 가능성 및 복잡성에 대해서도 고려해야 한다. 어떤 한 특성을 만족하는 TRS를 정의할 때 만약 그 시스템의 복잡도가 높은 경우에는 복잡도를 줄일 수 있는 TRS를 정의하는 것을 고려해 볼 필요가 있다. Orthogonal TRS와 다음 절에 논의할 call-by-need 계산법의 이론적 접근 방법을 위해서는 [7]을 참조하기 바란다.

2.3 전략(Strategies)

다음과 같은 룰을 생각해 보자 : A → A, F(x, 1) → 2 이고, 텀 F(A, 1)이 주어졌다고 하자. 이 텀은 두 개의 레덱스 A와 F(A, 1)을 포함하고 있다. 이 텀에 대한 정규형을 구하기 위해서 A에 대한 정규형을 먼저 구한 다음 F(A, 1)을 계산한다면 우리는 영원히 이 텀에 대한 정규형을 구할 수 없다. 그러나 F(A, 1)을 선택한다면 단 한 번의 rewriting으로 정규형에 도달할 수 있다. 예와 같이 계산할 레덱스의 선택 방법에 따라 정규형을 구할 수도 있고, 그렇지 못할 수도 있다. 이와 같이 계산할 레덱스를 선택하는 방법을 전략(strategy)라고 부르는 데, 전략은 텀과 텀 사이의 사상(mapping)으로서 볼 수 있다. 전략은 주로 주어진 텀의 정규형을 계산할 목적으로 정의되므로, 우리의 주된 관심은 정규형 전략에 모아진다. 문제는 원하는 전략이 존재할 때 "그 전략에 대한 사상을 순환함수(recursive function)로서 정의할 수 있는 가?"에 달려 있다. 그렇게 할 수 있다면 그 전략은 알고리즘으로서 표현할 수 있으며(computable), 따라서 자동화될 수 있어 프로그래머가 계산 과정(how)을 정의하지 않더라도 기계가 처리할 수 있다. 함수 프로그래밍에서 'what'만을 기술하고 'how'를 기술할 필요가 없다는 것은 바로 이러한 계산 가능한 전략이 존재하기 때문에 가능하다.

전략은 TRS의 수행 시간에 적용되므로 가능한 한 전략에 걸리는 시간이 짧도록 단순하고 신속하게 정의될 필요가 있다. 주어진 TRS에 대하여 어떤 전략이 정규형 전략인지는, 그리고 그것이 단순하게 정의될 수 있는 지는 TRS의 모습에 달려 있다. 다음의 두 전략에 대해서 생각해 보자.

Rules :

And(True, x) → x Or(True, x) → True
And(False, x) → False Or(False, x) → x

주어진 룰을 적용하여 다음 텀을 계산해 보자 :

And(And(True, And(True, False)), Or(False, True))

이 텀에 대하여 다음의 두 전략을 적용하여 선택되는 레덱스를 [ ]로 표시하면 다음과 같다.

(1) Leftmost-outermost : And([And(True, And(True, False))], Or(False, True))
(2) Parallel-outermost : And([And(True, And(True, False)])], [Or(False, True)])

(1)은 맨 바깥쪽의 레덱스 중에서 가장 왼쪽의 레덱스를 선택한 결과이며, (2)는 맨 바깥쪽의 모든 레덱스를 모두 선택한 것이다. Orthogonal TRS인 경우 (2)는 정규형 전략이다. (1)은 단 하나의 레덱스가 선택되는 데 비하여 (2)에서는 여러 개의 레덱스가 선택될 수 있다. (2)의 전략에서는 선택된 모든 레덱스들이 동시에 rewrite되어야 하므로 이것을 실제 구현하는 데 사용하기는 어렵다. 그러나 기억해야 할 것은 병렬 계산의 필요성이 시스템의 성능 때문이 아니라 원하는 계산을 정확하게 하기 위해서도 필요할 수 있다는 점이다. (2)는 multi-step 전략, (1)이 one-step 전략 혹은 순차적(sequential) 전략이라고 한다.

한 TRS가 순차적 전략을 갖는다는 것은 매우 바람직하다. 순차적 전략에 대하여 좀 더 구체적으로 논의해 보자. 우리가 원하는 순차

적 전략은 단순히 정규형을 구하는 것 뿐만 아니라 될수록 효과적이어야 한다. 만약 목적이 단순히 정규형만을 구하는 것이라면 [8]이 보여주는 바와 같은 모든 orthogonal TRS는 순차적 정규형 전략을 갖는다. 그러나 한편으로 생각해야 할 사항은 이러한 전략이 적용되는 경우에 정규형을 구하는 데 '반드시 rewrite되어야 하는 레덱스' 들만이 선택되는 것이 아니라는 점이다. 즉, 정규형에 도달한 rewrite들을 역추적 해보면 정규형을 구하기 위해서 rewrite되지 않아도 좋았을 레덱스들이 선택되는 것을 알 수 있다. 어떤 한 텀 t의 레덱스 r이 있을 때, r을 rewrite하지 않고서는 t의 정규형에 도달할 수 없다면 r은 t의 요구되는 레덱스 (needed redex)라고 한다. 모든 orthogonal TRS가 순차적 정규형 전략을 갖는다는 것은 rewrite 수행시 요구되는 레덱스만을 rewrite하면서 정규형에 도달한다는 뜻은 아니다.

요구되는 레덱스만을 선택하여 rewrite하는 순차적 정규형 전략을 갖는 TRS를 순차 TRS라고 부른다. 일반적으로 orthogonal TRS는 순차 TRS가 아니다. 한편, 순차 TRS를 결정하는 것은 모든 룰의 왼쪽과 오른쪽 텀 사이의 종속 관계를 전부 고려해야 하므로 불가능하다. 따라서, 순차 TRS 중에서 결정 가능한 TRS의 정의가 요구되고 있다. 순차 TRS 중에서 룰의 오른쪽 텀은 고려하지 않고 왼쪽만을 고려하여 결정될 수 있은 TRS를 강력 순차 (strongly sequential) TRS라고 부른다. 예를 들어, F(A, B, x) → 1, F(B, x, A) → 1, F(x, A, B) → 1의 세 룰로 이루어진 TRS는 강력 순차 TRS가 아니다. 한 텀 F(Redex1, Redex2, Redex3)가 주어졌을 때 이 세 레덱스 중에서 어떤 레덱스가 요구되는 레덱스인지를 룰의 왼쪽만을 보고서는 결정할 수 없기 때문이다. 이 예는 orthogonal이므로, orthogonal TRS가 강력 순차 TRS가 될 수 없음을 보여주고 있다. 그러나, 위의 룰을 조금 변형하여, 예를 들어, F(A, B, x) → 1, F(B, x, A) → 1의 두 룰로만 TRS가 정의되었다면, 텀 F(Redex1, Redex2, Redex3)에서 Redex1이 요구되는 레덱스임을 알 수 있다. 이 경우 TRS는 강력 순차 TRS가 된다. 강력 순차 TRS에서는 각 rewrite 단계마다 요구되는 레덱스를 결정할 수 있다. TRS를 함수 언어의 동작 원리 (operational semantics)로서 생각할 때 함수 언어는 강력 순차 TRS의 특수한 경우로서 간주할 수 있다[9].

3. 람다 계산법(Lambda Calculus)

람다 계산법은 앞서 논의한 TRS의 계산법과 여러 유사한 점이 많다. 본 절에서는 앞서 논의한 TRS의 특성과 비교하여 논의한다. 제공된 지면이 제한되어 있어 람다 계산법의 구체적인 논의를 할 수 없으므로 자세한 것은 [4]를 참조하기 바란다.

람다 계산법에서 사용되는 기호는 변수들과 λ 뿐이다. 람다 텀 또한 매우 간단하게 정의된다. 텀은 변수이거나, 한 텀 M이 주어졌을 때 λx.M (abstraction), 두 텀 M과 N이 주어졌을 때 (MN) (application) 또한 람다 텀이다. (λx.M)N의 형식을 갖는 텀은 (β)레덱스라고 한다. 이 레덱스는 다음과 같이 rewrite된다 : (λx.M)N → M[N/x]. 여기서 M[N/x]은 M의 모든 자유 (free) 변수 x를 N으로 대체한 결과를 의미한다. 이 rewrite는 일반적으로 문맥하에서도 성립한다 : C[(λx.M)N] → C[M[N/x]]. 람다 계산법에서 어떤 텀들은 정규형을 갖지 않는다. 예를 들어, (λx.xx)(λx.xx) → (λx.xx)(λx.xx) → ... . 람다 계산법에서 leftmost-outermost 전략은 정규형 전략이다. 즉, 한 텀이 정규형을 갖는다면 그 정규형은 leftmost-outermost 전략에 의해서 rewrite 하여 구할 수 있다. 예를 들어, (λa.λb. b)((λx. xx)(λx.xx)) → (λb. b). 만약 이 텀에 innermost 레덱스를 선택하는 전략을 적용한다면 이 전략으로는 정규형을 구할 수 없음을 쉽게 알 수 있다. 또한 람다 계산법은 Church-Rosser의 특성을 만족시킨다.

람다 계산법에서 λ는 하나의 인수만을 받아들이는 구조적 특징을 가지고 있다. 이것은 TRS 에서 소개된 여러 종류의 함수 기호들과는 대조적이며, 여러 인수를 입력으로 받는 함수가 어떻게 람다 계산법에서 표현될 수 있는지에 대하여 의문을 갖게 한다. 우선 우리가

인식해야 할 것은 람다 계산법에서는 기본적으로 모든 것이 함수로서 표현된다는 점이다. 이러한 인식은 우리가 편의상 함수를 함수와 인수 (혹은 프로그램과 데이터)로서 구별하는 인식과는 다소 차이가 있다. 따라서 함수 Plus가 두 개의 인수를 갖는다는 생각 대신, Plus는 (람다 계산법으로서 표현될 수 있는) 함수이며, (Plus A), (Plus A B), (Plus A B C ... K) 또한 람다 계산법의 정당한 구문적 표현으로서 보는 것이다. 따라서 인수로 전달되는 것은 반드시 (Plus A B) 형태로 될 필요가 없으며 Plus 로서도 전달될 수 있다. 이와 같은 개념에 의해서 표현되는 함수를 고차원적 함수 (higher order function)이라고 하며, 이러한 함수의 구문적 표현을 Currying이라고 한다.

앞서 말한 바대로 람다 계산법의 leftmost-outermost 전략은 정규형 전략이다. 따라서 이러한 전략을 람다 계산법의 구현에 적용하려고 할 것이다. 그러나 이때 고려해야 하는 점은 이 전략이 효율적이지 않을 수 있다는 점이다. 예를 들어, (λa. ... a ... a ...)와 같이 여러 개의 a가 바인드되어 있고, 텀 M이 매우 크고 많은 레덱스를 포함하였다고 가정하자. 이때 (λa. ... a ... a ...)M을 leftmost-outermost 전략에 의해서 계산한다면 (...M ... M ...)과 같이 여러 개의 M이 복사되어 결국 많은 레덱스를 포함하는 텀으로 rewrite되게 된다. 한편, 만약 M이 매우 작은 텀 M'을 정규형으로 갖는다고 하자. 그렇다면 (λa. ... a ... a ...)M → (λa. ... a ... a ...)M' → (...M' ... M')로 간단한 텀으로 rewrite될 수 있다. 전자의 방법은 프로그래밍 언어에서 call-by-name이라고 하고, 후자는 call-by-value라고 한다. call-by-value의 방법은 계산 결과가 공유될 수 있으므로 call-by-name에 비하여 훨씬 경제적이다. 그러나 call-by-value는 어떤 경우에는 call-by-name으로 계산할 수 있는 정규형을 찾지 못하는 단점이 있다. '정확한 계산법'과 '효율적인 계산법'을 선택해야 하는 고민이 있다. 그러나 실제로는 call-by-name의 비효율성을 해결할 수 있는 방법으로서 그래프 rewriting이 제안되어 [10] 실용적으로 이용되고 있다[1]. 위의 예에서 그래프 rewriting을 사용하면, M을 a 자리에 중복해서 복사하는 대신 모든 a가 어떤 하나의 위치를 포인트 하도록 하고 그 자리에 M을 위치시킨다. M이 rewrite되면 그 결과는 포인트하고 있는 모든 곳에서 공유할 수 있다. SML과 같은 언어에서는 call-by-value 기법을 사용하고 있고, Miranda™, Haskell, Clean과 같은 언어에서는 call-by-name 기법을 사용하고 있다.

람다 계산법이 소개되기 훨씬 전인 1924년에 Shönfinkel은 CL(Combinatory Logic)을 소개하였다. 후에 Curry는 Church의 람다 계산법과 CL 사이에 밀접한 관계가 있음을 밝혀내었다. CL은 람다 계산법과 유사하게 매우 간단한 세 종류의 기호 K, S, 변수를 사용하며, CL 텀은 변수, K, S, 혹은 두 CL 텀 P와 Q가 주어졌을 때 (P Q)로서 구성되어 있다. 일반적으로 K와 S에 I를 추가하여 이용하고 있다 (I = SKK로서 정의될 수 있다). 그리고 CL에는 세 룰 K P Q → P, S P Q R → P R (Q R), I P → P가 정의된다. TRS나 람다 계산법과 같이 레덱스를 포함하는 텀의 rewrite는 문맥을 사용하여 설명될 수 있다. 먼저, CL을 람다 계산법으로 표현하는 문제는 매우 쉽다. 변수는 그대로 이용하고 S = λa.λb.λc. ac(bc), K = λa.λb. a, I = λa. a로서 정의될 수 있다. 람다 텀을 CL 텀으로 변형하는 것은 다음과 같은 함수 ( )$_{CL}$ 정의된다:

$$(x)_{CL} = x,\ (MN)_{CL} = (M)CL(N)_{CL},$$
$$(\lambda x.M)_{CL} = \lambda^* x.(M)_{CL}$$

여기서 λ^*x.x = I, λ^*x.P = K P (x가 P에서 자유변수가 아닌 경우), λ^*x.PQ = S(λ^*x.P) (λ^*x.Q) 이다. 예를 들어 함수 Permute x y = y x 는 다음과 같이 표현될 수 있다. 우선 Permute는 람다 텀으로 Permute = λx.λy. y x 이다. 그러면 Permute은 CL 텀으로 (Permute)$_{CL}$ = S(K(SI))K이 된다. 이 변환은 (Permute)$_{CL}$ A B = S(K(SI))KAB → (K(SI)A)(KA)B → (SI)(KA)B → IB(KAB) → B(KAB) → BA 으로서 검증될 수 있다.

약간의 공리들이 추가되면 CL은 람다 계산법에 의해서 수행되는 모든 rewriting을 시뮬레이션 할 수 있다. 람다 계산법 (λx.M)N →

M[N/x]을 수행하기 위해서 M의 자유 변수 x를 찾는 것은 M이 크고 복잡한 형태를 갖는 경우 매우 긴 시간과 복잡한 구조를 관리해야 하는 문제를 갖고 있다. 수학적으로 이 계산이 기본 동작으로 정의되었을 지라도 프로그래밍 언어 측면에서 구현하는 데는 매우 복잡할 수 있다. 따라서 람다 계산법을(효과적으로) 구현하는 것은 어려운 일이었다. 위에서 소개한 변형 기법에 의하여 람다 계산법의 구현은 CL 텀으로 바꾸어 대신 계산할 수 있다. CL 텀으로 변형하는 경우 람다 계산법에서 요구되는 자유변수를 찾는 문제가 없어지는 유리한 점이 있다. 계산 가능성(computability)에서 잘 알려진 사실은 모든 계산 가능한 함수들이 람다 계산법으로 표현된다는 사실이다. 그런데, CL이 람다 계산법을 시뮬레이션 할 수 있으므로, 모든 계산 가능한 함수가 단 두 개의 기본 룰 S와 K로 표현될 수 있음을 보여준다. 이것은 매우 놀랍고도 중요한 사실이다. 이러한 기법은 실제 함수 언어 구현에서도 이용되고 있다. 함수 언어 Miranda™은 CL을 기반으로 정의된 SKIM(S.K.I. Machine)을 중간언어로 변환되어 수행된다. 이러한 기법은 수학에서 오랫동안 연구되어 왔던 결과들이 실용적으로 응용될 수 있음을 보여주는 좋은 예이다.

4. 연구 동향 및 결론

최근에 개발된 ML, Miranda™, Haskell. Clean 등의 언어는 그 기반 이론인 람다 계산법과 TRS의 개념을 충실히 반영하여 설계 구현되었다고 할 수 있다. 람다 계산법과 TRS는 이런 언어들의 설계와 구현에 밀접하게 연관되어 있다. 새로운 언어를 설계할 때나 기존 언어에 새로운 기능을 추가할 때, 그 특성들이 람다 계산법이나 TRS에서 이론적으로 검증될 수 있다면 그 언어의 타당성이 입증되었다고 볼 수 있다. 이러한 rewrite 시스템들의 문법 구조에서는 실제 사용되는 프로그래밍 언어보다 매우 간단하므로 이론적 검증이 쉽게 이루어질 수 있다. 또한, 이런 언어들은 람다 텀이나 TRS 텀으로 변환되어 구현되고 있다. 따라서 람다 계산법이나 TRS를 효율적으로 구현하는 기법은 함수 언어의 구현에 직접적으로 응용될 수 있다.

그래프 rewriting은 람다 계산법과 TRS에게 효율적인 계산법을 제공하고 있다. 그러나 그래프 rewriting은 무한(transfinite) 텀을 표현할 수 있을 뿐만 아니라 사용자에게 좀 더 친숙한 표현을 제공함으로써 TRS가 갖지 못하는 여러 중요한 기능을 가지고 있다. 따라서 이것은 TRS의 구현 기법이라기 보다는 그 자체의 연구 목적을 가지고 있는 중요한 분야이다. 컴퓨터 사이언스에서 함수의 개념이 중요한 역할을 하고 있으나 경우에 따라서 이것은 너무 추상화되어 있다고 보여진다. 벌써 1970년대에 수학에서는 정의된 적이 없는 프로세스(process)라는 개념이 컴퓨터 사이어스에서 연구되기 시작하여 그 동안 많은 결과가 소개되었다. 프로세스는 동시성 및 객체(agent 혹은 object 등)의 개념을 포함하고 있으며, 실제의 상황(real world)을 좀 더 자연스럽게 표현할 수 있게 한다. 프로세스에 대한 이론으로서 Petri-Net이 잘 알려져 있다. Pertri-Net이 단지 오토마타적인 특성만을 갖는 것에 비하여 Milner에 의해서 소개되고 있는 CCS와 Pi-calculus는 대수적(algebra) 특성을 함께 지니고 있으므로 동시성을 계산하는 데 매우 유익하며, 따라서 프로세스 대수(process algebra)라고도 불린다. 프로세스 대수의 연구는 비교적 짧은 역사를 가지고 있는 데, 그 동안 개발된 함수에 대한 이론이 프로세스 대수적인 측면에서 재검토되고 있다. 특히 Pi-calculus는 람다 계산법의 영향을 받아 만들어졌으며 여러 유사한 특성을 지나고 있다[11]. 이 분야에는 현재 많은 관심이 모아지고 있으며 앞으로 빠른 발전이 있을 것으로 예상된다.

타입 이론은 Church와 Curry에 의해서 오래 전에 제안되었는데, Hindley와 Milner가 좀 더 실용적으로 사용될 수 있는 이론을 제안하고 이것이 ML에서 성공적으로 구현되었다. 점차 함수 프로그래밍뿐만 아니라 객체지향 프로그래밍에서도 타입의 중요성이 강조되고 있다. 특히 최근 Girard에 의하여 제안된 Linear Logic (혹은 Linear type)은 이론적으로는 논리학 연구의 새로운 방향을 제시하고 있으며

[12], 프로그래밍 기법에서는 상태 제어, side-effect 프로그래밍, interactive 프로그래밍 기법을 정형화할 수 있는 이론으로서 받아들여지고 있다. 흥미로운 연구 결과가 예상된다.

주어진 지면이 제한되어 rewriting에 대한 많은 주요 결과들을 소개할 수 없었다. Rewriting은 프로그래밍 언어(특히 함수 프로그래밍) 및 정형방법(formal method)의 기반 기술로서 중요한 역할을 하고 있다. 함수 프로그래밍은 최근 매우 빠르게 성장하고 있으며 다목적 언어로서 사용될 수 있을 만큼 좋은 기능들을 갖추고 있다. 정형화 기법 또한 광범위한 분야에서 응용되고 있다. 지금까지 컴퓨터 관련 분야는 '사이언스'라는 용어가 어울리지 않을 만큼 기반 이론에 대하여 관심이 적었으나(특히 국내의 경우), 적어도 rewriting 분야만큼은 이론과 실용적 측면에서 성공적인 발전을 거두고 있음을 강조하고자 한다.

참고문헌

[1] S. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

[2] N. Dershowitz and J.P. Jouannaud, "Rewrite System," In (J. van Leeuwen, ed.) *Handbook of Theoretical Computer Science*, vol. B, North-Holland, 1991. pp 243-320.

[3] J.W. Klop, "Term Rewriting Systems," In (S. Abramsky et al eds.) *Handbook of Logic in Compute Science*, vol. 2. Oxford University Press, 1992, pp. 1-116.

[4] H.P. Barendregt, *The Lambda Calculus, its Syntax and Semantics*, revised edition, North-Holland, 1984.

[5] J.R. Hindley and J.P. Seldin, *Introduction to Combinators and Lambda Calculus*, Cambridge University Press, 1986.

[6] H.P. Barendregt et al, "Term Graph Rewriting," In (J.W.deBakker, et al eds.) *PARLE Conference Proceedings*, vol 2. Lecture Notes in Computer Science 259, Springer-Verlag, 1987, pp. 141-158.

[7] G. Huet and J.-J. Levy, "Call-by-need Computation in Non-ambiguous Linear Term Rewriting Systems," In (J.L. Lassez and G. Plotkin eds.) *Computational Logic : Essays in Honour of Alan Robinsons'*, MIT Press, 1991, pp 394-443.

[8] J.R. Kennaway, "Sequential Evaluation Strategies for Parallel-Or and Related Reduction Systems," *Annals of Pure and Applied Logic*, 43, 1989, pp. 31-56.

[9] Y. Toyama et al, "The Functional Strategy and Transitive Term Rewriting Systems," In (M.R. Sleep and M.J. Plasmeijer, and M. C.J.D. van Eekelen eds.) *Term Graph Rewriting Theory and Practice*, John Wiley & Sons, 1993, pp. 61-75.

[10] C.P. Wadsworth, *Semantics and Pragmatics of the Lambda Calculus*, D. Phil. thesis, Programming Research Group, Oxford University, 1971.

[11] R. Milner, "The Polyadic π-calculus : a Tutorial," In (F.L. Bauer et al eds.) *Logic and Algebra of Specification*, Springer-Verlag, 1993.

[12] J.-Y. Girard, "Linear Logic : Its Syntax and Semantics," *Theoretical Computer Science*, 50, 1987, pp 1-102.

변 석 우

1980 숭실대학교 전자계산학과 졸업(학사)
1982 숭실대학교 대학원 전자계산학과 졸업(석사)
1982~현재 한국전자통신연구소 병렬프로그래밍 연구실
1989~94 영국 East Anglia 대학(전산학 박사)
관심분야 : Lambda Calculus와 Term Rewriting, Linear Logic, Type Theory, Semantics, Process Algebra, FunctionalProgramming, 병렬/분산 프로그래밍