상태를 갖는 함수형 프로그래밍 언어의 수행모델

주 형 석† · 김 홍 읍†† · 유 원 희†††

요 약

순수 함수형 프로그래밍 언어는 명확한 어의와 많은 특성에도 불구하고 상태의 표현이 어렵다는 문제점을 가지고 있다. 이와 같은 문제로 인하여 순수 함수형 언어에 상태를 표현하기 위한 많은 연구가 진행되었으나 형 시스템이나 감축규칙이 복잡하기 때문에 구현이 어렵다는 문제점이 발생된다. 따라서 효율적인 구현을 위하여 감축규칙을 단순화시키고 상태를 효율적으로 관리할 수 있는 방안이 요구된다.

본 논문에서는 순수 함수형 언어의 성질을 침해함이 없이 상태를 표현할 수 있으며 감축규칙을 단순화시킨 함수형 언어의 수행모델 st-계산을 제안하고, 제안된 모델이 Church-Rosser 정리를 만족함을 증명하였다. 제안된 방법을 통해 구문구조의 표현력을 높일 수 있었으며, 감축규칙을 단순화함으로써 구현이 용이할 것으로 기대된다.

Execution Model for Functional Programming Language with States

Hyung Seok Joo† · Hong Eub Kim†† · Weon Hee Yoo†††

ABSTRACT

Despite elegant semantics and a lot of features, pure functional programming languages do not provide an efficient way of representing states. Many researches have been done to resolve the problem, however, another problem arises that it is hard to implement because of the complex type system and reduction rule. Threrefore, the scheme which simplifies the reduction rule and maintains states efficiently is needed to have the implementation effective.

This paper proposes st-calculus, the execution model of a functional language with states, and proves that the proposed model satisfies the Church-Rosser theorem. It has simple reduction rules and the ability of representing states without compromizing the properties of pure functional languages. The expressiveness can be increased through this model, and the difficulties with implementation may be reduced by simplifying the reduction rules.

1. 서 론

컴퓨터 프로그래밍 언어 및 계산모델 분야에서 연구되고 있는 함수형 언어는 고계함수, 지연평가, 참조적 투명성, 자료 추상화, 패턴매칭 등의 여러가지 특징으로 인하여 커다란 관심의 대상이 되고 있다. 이와 같은 함수형 언어는 수학적 함수에 기반을 둔 적용형 언어(applicative language)로서 λ-계산(λ-calculus)을 기본으로 하고 있으며, 프로그램의 전체적인 구조가 함수호출 관계로 구성되어 있다[1, 11].

참조적 투명성이 보장되는 함수형 언어는 명령형 언어(imperative language)와는 달리 부작용(side effect)

※본 논문의 연구는 1995년도 인하대학교 교내연구비와 정보통신부 96년도 대학 기초연구 지원사업(과제번호:96109-ITI-I1)의 지원에 의해 수행됨.

† 정 회 원:유한전문대학 전자계산과 부교수
†† 정 회 원:인하대학교 전자계산공학과 박사과정
††† 종신회원:인하대학교 전자계산공학과 교수

논문접수:1996년 8월 16일, 심사완료:1996년 11월 22일

이 존재하지 않으므로 프로그램의 이해와 검증이 용이하며, 묵시적인 병렬성(implicit parallelism)을 자연스럽게 지원한다. 그러나 함수형 언어에서는 상태(state)를 갖는 알고리즘을 나타내기가 부자연스럽고, 입출력이나 동적 자료구조의 처리가 어렵다는 점이 지적되고 있다[3, 6, 19].

함수형 언어의 대수적 성질을 침해하지 않으면서 상태를 표현하기 위한 대표적인 연구로는 Moggi[15]에 의해 제안된 monad를 기반으로 한 연구[18, 23, 24], Swarup의 ILC[22], Odersky의 λ_{var} 계산모델[16] 등이 있다. monad를 이용한 방법은 상태 정의가 집중화 되는 문제점을 가지며, ILC에서는 재귀적 개념이 배제되어 있으며 구문구조가 3-단계의 형 시스템(type system)에 의해 정의되므로 상태의 표현이 제한된다. 또한 λ_{var} 계산모델에서는 감축규칙이 복잡하며 엄격한 구문구조를 적용하기 때문에 제한된 표현에 의해 프로그램을 구성하여야 하며, 구현이 용이하지 않다는 단점을 갖는다.

본 논문에서는 순수 함수형 언어의 대수적 성질을 침해함이 없이 상태를 표현할 수 있으며, 구문의 표현력을 높이고, 감축규칙을 단순화한 함수형 언어의 계산모델 λ_{st}-계산을 설계하고자 한다. 또한 λ_{st}-계산이 함수형 언어의 대수적 성질을 만족함을 보인다.

본 논문의 구성은 다음과 같다. 2장에서 상태를 표현함에 있어서 나타나는 부작용을 처리하기 위한 방법들을 고찰하고, 3장에서는 본 논문에서 관점으로 하는 사항들을 바탕으로 구문구조와 감축규칙을 정의하고 적용 예를 나타낸다. 4장에서는 정의된 모델에 대한 대수적 성질을 검증하며, 5장에서 본 논문의 결론을 제시한다.

2. 관련연구

컴퓨터 프로그래밍 언어 및 계산모델 분야에서 연구되고 있는 함수형 언어는 여러가지 특징으로 인하여 커다란 관심의 대상이 되고 있다. 함수형 언어가 지닌 특징 중, 특히 주목할 만한 것은 명령형 언어와는 달리 부작용이 존재하지 않으므로 참조적 투명성이 보장되어 프로그램의 이해와 검증이 용이하고 묵시적인 병렬성을 자연스럽게 지원한다는 것이다[12, 13, 14].

참조적 투명성은 부작용이 발생되지 않도록 함으로써 보장될 수 있는데, 부작용의 발생은 명령형 언어에서 상태를 변화시키는 배정문에 의해 하나의 상태가 임의적으로 변화하는데서 기인한다. 따라서, 함수형 언어에서는 참조적 투명성을 보장하기 위하여 명령형 언어에서 사용하는 배정문을 허용하지 않고 함수호출에 의하여 값을 생성하도록 함으로써 각 상태가 임의적으로 변화되지 못하도록 하고 있다. 이와 같이 함수형 언어에서는 임의적으로 상태를 변화시키는 배정문을 제거함으로써 수식의 계산 순서에 구애됨이 없이 자연스럽게 묵시적 병렬성을 지원하는 장점을 가지고 있다. 반면에, 다음과 같은 몇가지 사항에 대해서는 명령형 언어에 비하여 어려운 점을 초래하고 있다[3, 6, 19].

첫째, 알고리즘을 표현하는데 있어서 어려움이 있다.
둘째, 프로그램의 입/출력을 기술하는데 어려움이 있다.
세째, 동적 자료구조의 처리에 어려움이 있다.

함수형 언어에서의 이러한 문제점을 극복하기 위하여 비순수 함수형 언어인 Standard ML, Scheme 등에서는 제한된 배정문의 사용을 허용하고 있다. 또한 함수형 언어와 명령형 언어의 특성을 결합하기 위한 시도[3, 17]가 진행되었다. 그러나, 이들 언어에서는 프로그래밍의 편의성을 향상시키기 위하여 배정문을 허용하면서 그로 인해 초래될 수 있는 부작용의 발생 가능성을 배제하지 않고 있다. 그 결과 프로그래밍의 편의성은 향상시킬 수 있었으나, 부작용의 발생으로 인하여 참조적 투명성이 파괴되어 수식의 계산 순서가 프로그램 전체의 수행결과에 영향을 미치게 된다[9, 10]. 이렇게 함으로써 순수 함수형 언어가 지니는 커다란 특징인 묵시적 병렬성을 저해하게 되어, 함수형 언어가 병렬형 계산모델에 효율적으로 활용될 수 있다는 큰 특성에 부정적인 영향을 주게 된다.

부작용의 발생으로 인해 야기되는 참조적 투명성의 문제는 함수형 언어에서 만의 문제로 국한되지 않고 명령형 언어 측면에서도 중요한 논점의 하나가 되고 있다. Algol 60을 시점으로 하여 명령형 언어 측면에서 부작용을 다루기 위한 연구가 많이 진행되었다. 그 결과 대표적인 언어로서 Reynolds의 Forsythe[20], Felleisen의 λ-V-CS calculus[5], Gifford의 Effect Sys-

tem[7] 등이 제안되었다. 그러나 이와 같은 언어들은 순수 함수형 언어에 상태를 도입하기 위한 시도는 아니었다[16, 21].

함수형 언어에서도 상태를 표현하기 위한 연구들이 많이 진행되고 있다. 부작용의 발생을 초래하지 않는 범위에서 함수형 언어에 상태를 도입하는 단순한 방법으로는 상태를 명시적으로 표현하는 방법[4, 8]을 들 수 있다. 그러나, 이 방법에서는 명령형 언어에서 동적 자료가 처리될 때 얻어지는 것과 같은 효과를 얻지 못하고 있다. 즉, 공유된 동적 자료를 변경하였을 경우 해당 동적 자료를 공유하고 있는 모든 곳에 자동으로 영향을 줄 수 있어야 하는데, 이 방법에서는 전체 자료 구조의 복사본을 생성하여 공유하는 곳에 명시적으로 전달해야 하는 문제점이 따른다. 이러한 문제점을 해결하기 위해 제안된 monad를 이용한 방법[23, 24]은 상태 인자(state parameter)를 명시적으로 표현해야 하는 오버헤드는 없앴으나 상태 정의가 집중화 된다는 문제점과 동적 자료의 처리가 어렵다는 점이 지적되고 있다[16, 22].

함수형 언어의 대수적 성질을 침해하지 않으면서 명령형 프로그래밍 언어의 사양을 확장하기 위한 대표적인 연구로는 Swarup에 의해 제안된 ILC(Imperative Lambda Calculus)[22]와 Odersky에 의해 제안된 이름 호출에 기반을 둔 λ_{var} 계산모델[16]이 있다. ILC에서는 상태 중심의 계산을 추상화한 형태인 *observer*에 의해 상태의 값을 얻는 반면에 λ_{var} 계산모델에서는 상태 변환자를 이용하여 상태로부터 값-상태의 짝을 얻는다.

이와 같이 ILC와 λ_{var} 계산모델은 함수형 언어의 기본 속성을 침해하지 않으면서 명령형 언어의 사양을 확장하고 있다. 그러나 ILC는 재귀적 개념이 배제되어 있으며 3-단계의 형 시스템에 의해 정의되는데, 이로 인하여 상태의 표현이 제한을 받게 된다는 문제점을 갖고 있다. λ_{var} 계산모델은 명확한 어의를 갖고 있지만 상태의 정확한 표현을 위해서는 감축규칙의 정확한 사용을 요구한다는 것과 감축규칙이 복잡하고 감축과정에서 프로그램의 동적 변환이 많이 요구되므로 구현이 용이하지 못하다는 문제점이 존재한다. 예를 들어, λ_{var} 계산 모델에서 $N=:v;M_1;var\ w.\ v? \triangleright x.M_2;M_3$와 같은 프로그램은 순차적으로 다음과 같은 과정으로 감축되어 진다.

$N=:v;M_1;var\ w.\ v? \triangleright x.M_2;M_3 \rightarrow b_2$
$N=:v;M_1;v? \triangleright x.\ var\ w.\ M_2;M_3 \rightarrow b_1$
$N=:v;v? \triangleright x.\ M_1;var\ w.\ M_2;M_3 \rightarrow f$
$N=:v;(x.\ M_1;var\ w.\ M_2)\ N;M_3$
… … …

이러한 감축과정에서 특히, b_1에 의한 감축은 상태변수 v가 의미하는 값을 결정하기 위하여 v?을 v에 대한 배정문을 처음으로 만나는 위치까지 이동시켜서 감축을 수행한다. 즉, 해당되는 표현을 대응되는 수식의 위치로 이동시키기 위하여 프로그램을 동적으로 재구축해야만 하는 경우가 발생된다. 그리고, 여기서 N=:v는 저장의 개념이 없는 배정 연산이므로 프로그램의 나머지 부분에 존재할 수 있는 또 다른 v?이 b_2 감축에 의해 N=:v가 참조할 수 있도록 f에 의한 감축시 v에 대한 배정 연산을 그대로 보존한 상태에서 v가 의미하는 값인 N을 ▷에 의해 합성되는 함수의 인자로 전달해야 한다. 이러한 이유로 이론적인 면에서는 λ_{var} 계산 모델이 함수 언어의 속성을 최대한 따르고 있다고 할 수 있으나 구현상에 어려움을 보이고 있으며, 감축규칙이 복잡해 진다.

3. 구문구조 및 감축규칙의 정의

3.1 λ_{st}-계산의 구문구조

프로그래밍의 표현력과 편의성을 향상하기 위하여 함수형 언어에 상태를 표현하고 조작할 수 있도록 정의한 λ_{st}-계산의 구문구조는 〈표 1〉과 같다. λ_{st}-계산은 확장-계산에 상태를 표현하기 위한 구조를 확장한 계산모델이다.

〈표 1〉 λ_{st}-계산의 구문구조
〈Table 1〉 Syntax of λ_{st}-calculus

$E ::= A \mid I \mid E_1 \parallel E_2 \mid E_1 \oplus E_2$
$A ::= c \mid x \mid f \mid \lambda x.E \mid A_1\ A_2 \mid$ letrec $\{x = E\}^+$ in E
$I ::= \lambda_{st} v.E \mid v \mid T \triangleright \lambda x.E \mid [v_1\ v_2 \ldots v_n]? \triangleright \lambda x_1.\lambda x_2. \ldots .\lambda x_n.E \mid v:=A$
$T ::=$ st E | app E | v?

본 논문에서 정의된 st-계산의 구문구조는 순수 λ-항, 상태변수(mutable variable)를 정의하고 상태를 관

리하기 위한 상태 관리 연산자(state transfomer), 상태 합성 연산자(state composition) 등으로 구분된다. 〈표 1〉에서 보인 λ_{st}-계산의 주요 구문은 다음과 같은 의미를 갖는다.

■순수 λ-항:순수 λ-항에서는 확장 λ-계산에서 사용되는 구문이 어의의 변화없이 그대로 사용된다. 〈표 1〉에서 A로 표현된 λ-항들이 이에 해당되는 것으로서, 기본형의 상수값 c, 한정변수 x, δ-감축규칙에 의해 수행되는 +, −, *, /, 조건문 등의 원시 연산자 f, 함수 추상화 λx.E, 순수 λ-항들의 함수적용을 나타내는 A_1 A_2, 재귀구조를 표현하기 위한 letrec $\{x=E\}^+$ in E 등이 있다.

■상태변수의 정의와 상태 관리 연산자:λ_{st}-항 내에서 상태를 표현하기 위해서는 상태변수를 먼저 정의하여야 하는데 이는 λ_{st} v.E에 의해 이루어 진다. λ_{st} v.E에서 상태변수는 v이며, v의 영역은 식 E로 제한된다. 그리고, λ_{st}-계산에서 정의된 상태변수를 관리하는 상태 관리 연산자는 상태의 값을 참조하기 위한 ?, 상태를 변경하기 위한 :=, 다른 함수 추상화에 전달될 인자를 생성하는 st, app 등이 있다.

상태 관리 연산자 ?, :=는 스트릭트성(strict) 인자를 가지는 연산자이다. v?은 현재의 상태에 영향을 주지 않고 상태변수 v의 값을 참조하기 위한 상태 관리 연산자로서 ?이 상태 합성 연산자 ▷과 결합되면 참조된 값을 ▷에 의해 합성될 함수의 인자로 전달한다. v:=A는 식 A의 계산 결과를 상태변수 v에 저장하는 것으로서 :=의 LHS(left hand side)는 상태변수이며 RHS(right hand side)는 λ_{st}-항으로 구성된다. 상태 변환자 ?과 :=에 관한 몇가지 예와 의미는 다음과 같다.

- n:=m? m의 값을 참조하여 상태변수 n에 저장
- n:=m?+10 m?+10의 계산 결과를 상태변수 n에 저장
- v1:=v2 상태 관리 연산자 :=의 형 오류 (type error)

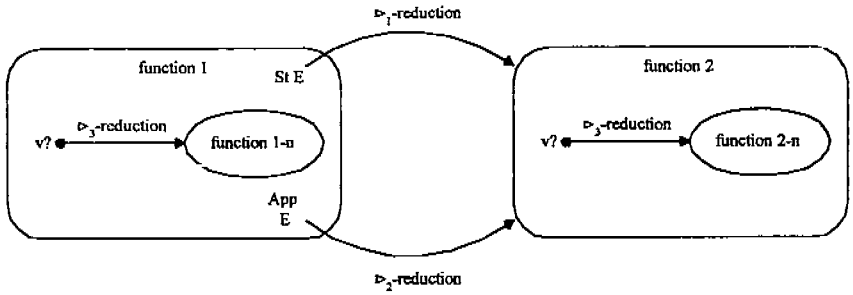
또한, st와 app는 상태변수의 영역을 확장하기 위한 것으로서 st는 상태 합성 연산자 ▷과 결합되어 현재의 함수내에서 사용되고 있는 상태의 전부를 지정된 함수로 전달하며, app는 현재의 최종 상태를 다른 함수에 전달하기 위한 것으로서 함수내에서 정의된 상태를 파괴하면서 영역을 확장한다.

- st A λ_{st}-항 st A가 포함된 함수에서 정의된 모든 상태의 영역을 합성될 함수로 확장하고 A를 합성될 함수의 인자로 전달.
- app I λ_{st}-항 I를 정규형으로 계산하고 app I가 포함된 함수내에서 정의된 모든 상태를 파괴한 후 I의 정규형을 합성될 함수의 인자로 전달.

■상태 합성 연산자:상태식 합성이란 어떤 함수에서 정의된 임의의 상태식을 지정된 함수 추상화로 전달하는 것으로서 전달할 상태식을 지정된 함수 추상화의 인자로 전달하여 β-감축식을 생성한다. 이러한 상태식 전달은 상태 합성 연산자 ▷에 의해 표현되며, ▷에 의해 지정된 함수에 인자로 전달될 상태식은 ?, st, app 등의 상태 관리 연산자에 의해 생성된다. (그림 1)은 본 논문에서 정의한 λ_{st}-계산에서의 상태 관리 연산자와 상태식 합성 연산자의 관계에 의해 발생되는 상태식의 합성 개념을 표현한 것이다.

(그림 1)에서 $\triangleright_1$, $\triangleright_2$로 표현된 상태 합성 연산자는 함수와 함수사이에서 발생하는 함수 적용으로서 함수1내의 임의의 상태식을 함수2의 인자로 합성하여 β-감축을 유도한다는 공통점을 가지고 있다. 반면, $\triangleright_1$, $\triangleright_2$의 차이점은 다음과 같은 두가지로 정의된다. 첫째, $\triangleright_1$은 함수1내의 상태 관리 연산자 st에 의해 생성되는 상태식을 함수2의 인자로 전달하면서 함수1내에서 정의된 현재의 모든 상태까지 그대로 상속시켜 주는 반면에, $\triangleright_2$는 함수1내의 상태 관리 연산자 app에 의해 생성되는 상태식을 함수2의 인자로 합성한 후 함수1에서 정의된 모든 상태를 파괴한다. 둘째, st $\triangleright_1$ 상태식 합성에서 st는 비스트릭트성(non-strict) 인자를 취하지만 app $\triangleright_2$ 상태식 합성에서의 app는 스트릭트성 인자를 취하게 된다.

한편, ▷3은 어떤 함수내에서 정의된 상태변수의 값을 상태 관리 연산자 ?에 의해 생성하여 그 자신에 포함된 부함수의 인자로 전달하는 상태 합성 연산자이다. 그리고, v? $\triangleright_3$ λx.E와 같은 상태식 합성 연산에서 프로그래밍의 편의성을 향상시키기 위한 구문구조로서 n개의 상태변수의 값을 n개의 한정변수로 구성되는 함수 추상화에 합성하는 [v_1 v_2 ... v_n]? $\triangleright_3$ λx_1.



(그림 1) 상태식의 합성 개념도
(Fig. 1) Composition diagram of state expression

λx_2. .... .λx_n.E와 같은 구조를 정의한다. (그림 1)의 개념에 따른 상태 합성 연산자를 사용한 λ_{st}-항의 구성 형태를 보이면 다음과 같다.

- (...st E1) ▷ (λx. E)
- (...app E1) ▷ (λx. E)
- (...v? ▷ λx. E)
- (...[v_1 v_2 ... v_n]? ▷ λx_1.λx_2. .... .λx_n.E)

■확장 원시 연산자(extended primitive operator) :λ_{st}-계산에서 상태를 관리하게 되면서 참조적 투명성 보장과 프로그래밍의 편의성을 향상시키기 위해서는 λ_{st}-항의 실행순서를 제어할 필요가 있다. 이를 위하여 λ_{st}-계산에 실행순서를 제어하기 위하여 ‖, 의 이항 원시 연산자(binary primitive operator)를 정의한다. ‖는 피연산자1과 피연산자2 사이에 수행순서가 없는 병행 관계를 표현하며, ⊕는 피연산자1을 먼저 수행한 후 피연산자2를 수행해야할 수식을 표현하며, 피연산자1에 의한 상태 변화를 피연산자2에서 활용하는 경우에 사용된다. ‖, ⊕의 구성 예와 감축의 의미를 나타내면 다음과 같다. 다음에서 $EVAL$(E)는 E를 계산한 결과를 의미하며, λx.E에서 x $\notin fv$ E이다.

- $E_1 \| E_2 \Rightarrow \begin{cases} (\lambda x.\ E_1)\ EVAL(E_2) \\ (\lambda x.\ E_2)\ EVAL(E_1) \end{cases}$
- $E_1 \oplus E_1 \Rightarrow (\lambda x.\ E_2)\ EVAL(E_1)$

3.2 λ_{st}-계산의 감축규칙

λ_{st}-항을 계산하는데 사용되는 λ_{st}-계산의 감축규칙은 다음 〈표 2〉와 같다. 〈표 2〉에서 EVAL(E)는 항 E를 계산한 결과를 나타내며, fv(E)는 항 E에서 자유변수의 집합을 의미한다. 또한, 제안된 감축규칙에서는 해당 λ_{st}-항에 대한 상태 집합을 나타낸다.

λ_{st}-계산에서의 감축에는 β-감축, δ-감축, 확장된 감축 등이 있다. β-감축과 δ-감축은 확장 λ-계산에서 적용되는 감축과 동일하며, 확장된 감축은 λ_{st}-계산이 정의되면서 도입된 상태와 관련하여 정의된 감축규칙이다.

λ_{st}v. E는 영역이 E로 제한되는 값이 정의되지 않은 상태변수 v를 생성하게 되며, v?는 상태변수 v의 현재 값을 참조하는 연산이다. 그리고, :=는 두 번째 인자를 스트릭트성 인자로 간주하여 계산한 결과를 첫 번째 인자에 해당하는 상태변수에 저장하는 연산이며, ‖와 ⊕는 상태가 도입됨으로써 발생하는 제어 문제를 〈표 2〉에서 주어진 것과 같은 순서로 처리하기 위하여 정의된 연산자이다. 그리고, st ▷, app ▷, ? ▷ 등은 3.1절에서 전술한 바와 같이 상태변수의 영역확장 및 상태식의 전달을 위하여 정의된 연산자이다.

3.3 λ_{st}-계산의 적용

본 절에서는 λ_{st}-계산에서 정의된 상태 관리 연산을 활용하여 작성된 프로그램의 적용 예를 보인다. 첫 번째 프로그램은 누산기 역할을 하는 상태변수를 정의하여 인자로 전달되는 값들의 누적합을 유지하면

〈표 2〉 st-계산의 감축규칙

〈Table 2〉 Reduction rule of st-calculus

- β-reduction : $(\lambda x.E_1)\,E_2 \rightarrow [E_2/x]\,E_1$
- δ-reduction : $f\ V \rightarrow \delta(f, V)$
- enriched reduction

λ_{st}	$:\lambda_{st} v.\ E$	ρ	$\rightarrow E\ \rho \cup \{\perp/v\}$
?	$:v?$	$\rho \cup \{N/v\}$	$\rightarrow N\ \rho \cup \{N/v\}$
:=	$:(v:=E_1)\ E_2$	$\rho \cup \{N/v\}$	$\rightarrow E_2\ \rho \cup \{EVAL(E_1)/v\}$
‖	$:E_1 \| E_2$	ρ	$\rightarrow \begin{cases} (\lambda x.E_2)\ EVAL(E_1)\ \rho ; x \notin fv(E_2) \\ (\lambda x.E_1)\ EVAL(E_2)\ \rho ; x \notin fv(E_1) \end{cases}$
$\oplus$	$:E_1 \oplus E_2$	ρ	$\rightarrow (\lambda x.E_2)\ EVAL(E_1)\quad \rho ; x \notin fv(E_2)$
st $\triangleright$	$:st\ A \triangleright (\lambda y.\ E)$	ρ	$\rightarrow (\lambda y.\ E)\ A \qquad \rho$
app $\triangleright$	$:app\ A \triangleright (\lambda y.\ E)$	ρ	$\rightarrow (\lambda y.\ E)\ EVAL(A)\quad \{\}$
? $\triangleright$	$:v? \triangleright (\lambda y.\ E)$	$\rho \cup \{N/v\}$	$\rightarrow (\lambda y.\ E)\ N \qquad \rho \cup \{N/v\}$

서 상태변수에 누적된 현재 값을 최종적으로 생성하며, 두 번째 프로그램은 첫 번째 프로그램과 마찬가지로 인자의 누적합을 구하지만 최종 결과가 첫 번째 프로그램과는 달리 상태변수의 현재 값이 아닌 이전 값이 생성되는 경우를 보인다. 즉, 두 번째 프로그램은 〈표 2〉의 st $\triangleright$ 감축에 의해 이전 상태까지도 관리될 수 있음을 보여준다. 그리고, 세 번째 프로그램은 재귀적 구조를 활용한 프로그램 예를 보인다.

■ 적용 1 : 누적합 생성 프로그램 accumulate1

누적합 생성 프로그램에 전달되는 인자 값들의 누적 결과를 생성하는 프로그램이다. 이를 위한 누적합 생성 프로그램은 상태변수의 정의를 포함하여 다음과 같이 작성할 수 있다.

- accumulate1 = $(\lambda init.\lambda_{st}\,cnt.\ (cnt:=init) \oplus (st\ \lambda inc.\ (cnt:=cnt?+inc) \oplus (cnt?)))$

이와 같이 작성된 accumulate1에 누적할 인자 1, 2를 적용하여 감축되는 과정을 보이면 다음과 같다. 〈표 2〉에서 정의된 감축규칙을 적용하여 프로그램을 감축하는 과정에 대한 표시를 위해 각 단계의 감축식을 밑줄을 사용하여 표시한다.

((accumulate1 0) $\triangleright$ (ctr. ctr 1 $\triangleright$ x. ctr 2)) $\quad \rho \equiv \{\}$

(($\underline{(\lambda init.\lambda_{st}\,cnt.(cnt:=init) \oplus (st\ \lambda inc.(cnt:=cnt?+inc) \oplus (cnt?)))\ 0}$) $\triangleright$ (λctr.ctr 1 $\triangleright$ λx.ctr 2)) $\quad \rho \equiv \{\}$

(($\underline{\lambda_{st}\,cnt.(cnt:=0) \oplus (st\ \lambda inc.(cnt:=cnt?+inc) \oplus (cnt?))}$) $\triangleright$ (λctr.ctr 1 $\triangleright$ λx.ctr 2)) $\quad \rho \equiv \{\perp/cnt\}$

(($\underline{(cnt:=0) \oplus (st\ \lambda inc.(cnt:=cnt?+inc) \oplus (cnt?))}$) $\triangleright$ (λctr.ctr 1 $\triangleright$ λx. ctr 2)) $\quad \rho \equiv \{0/cnt\}$

(($\underline{st\ \lambda inc.(cnt:=cnt?+inc) \oplus (cnt?)}$) $\triangleright$ (λctr.ctr 1 $\triangleright$ λx. ctr 2)) $\quad \rho \equiv \{0/cnt\}$

((λctr. ctr 1 $\triangleright$ λx.ctr 2) ($\underline{\lambda inc.(cnt:=cnt?+inc) \oplus (cnt?)}$)) $\quad \rho \equiv \{0/cnt\}$

(($\underline{\lambda inc.(cnt:=cnt?+inc) \oplus (cnt?))\ 1}$) $\triangleright$ (λx.(λinc.(cnt:=cnt?+inc) $\oplus$ (cnt?)) 2)) $\quad \rho \equiv \{0/cnt\}$

(($\underline{cnt:=cnt?+1}$) $\oplus$ (cnt?)) $\triangleright$ (λx.(λinc.(cnt:=cnt?+inc) $\oplus$ (cnt?)) 2)) $\quad \rho \equiv \{1/cnt\}$

(($\underline{cnt?}$) $\triangleright$ (λx.(λinc.(cnt:=cnt?+inc) $\oplus$ (cnt?)) 2)) $\quad \rho \equiv \{1/cnt\}$

(($\underline{\lambda x.(\lambda inc.(cnt:=cnt?+inc) \oplus (cnt?))\ 2}$) $\underline{1}$) $\rho \equiv \{1/cnt\}$

(($\underline{\lambda inc.(cnt:=cnt?+inc) \oplus (cnt?))\ 2}$) $\quad \rho \equiv \{1/cnt\}$

(($\underline{cnt:=cnt?+2}$) $\oplus$ (cnt?)) $\quad \rho \equiv \{3/cnt\}$

$\underline{cnt?} \quad \rho \equiv \{3/cnt\}$

3 $\quad \rho \equiv \{3/cnt\}$

■ 적용 2 : 누적합 생성 프로그램 accumulate2

누적합 생성 프로그램 accumulate2는 적용 1에서 보인 것과 같은 상태변수의 현재 상태에 대한 참조뿐만 아니라 st $\triangleright$에 의해 이전 상태를 참조하는 예를 보이는 프로그램이다. accumulate2의 구성과 이에 대한 적용과 결과는 다음과 같다.

· accumulate2 = (λinit.λ_{st}cnt.(cnt := init)
⊕ (st λinc.cnt? ▷ λ_c.(cnt := c + inc) ⊕ (st c)))
(((accumulate2) 0) ▷ (λctr.ctr 1 ▷ λx.ctr 2)) ▷ (λx.x)
...
1

■ 적용 3: 선형 탐색 프로그램 counter

선형 탐색 프로그램은 재귀적 구조를 보여주기 위한 것으로서 리스트상에서 특정 요소의 수를 상태변수에 누적하면서 계산하는 프로그램이다. 이를 위한 프로그램 구조는 다음과 같다. 선형 탐색 프로그램 counter는 리스트상에서 요소 '2'의 개수를 계산하는 프로그램이다. 적용 3에 대한 감축과정은 리스트의 첫 번째 요소를 판단하는 단계까지 만을 표현하였으며 나머지는 동일한 과정을 반복하게 된다.

· counter =
(λx.λ_{st}cnt.λ_{st}p.((cnt := 0)‖(p := x)) ⊕ (p? ▷ letrec count
= (λn.λ_{st}k.(k := head(n))
⊕ (if (k? = nil) then cnt?
else ((if (k? = 2) then cnt := cnt? + 1)
⊕ ((k := tail(n)) ⊕ (count k)))))))

(counter) [1:2:3:2:5]
(λx.λ_{st}cnt.λ_{st}p.((cnt := 0)‖(p := x)) ⊕ (p? ▷ letrec count
= (λn.λ_{st}k.(k := head(n))
⊕ (if (k? = nil) then cnt?
else ((if (k? = 2) then cnt := cnt? + 1)
⊕ ((k := tail(n)) ⊕ (count k)))))) [1:2:3:2:5]
ρ ≡ { }
(((cnt := 0)‖(p := [1:2:3:2:5])) ⊕ (p? ▷ letrec count
= (λn.λ_{st}k.(k := head(n))
⊕ (if (k? = nil) then cnt?
else ((if (k? = 2) then cnt := cnt? + 1)
⊕ ((k := tail(n)) ⊕ (count k)))))))
ρ ≡ {0/cnt, [1:2:3:2:5]/p}
(p? ▷ letrec count = (λn.λ_{st}k.(k := head(n))
⊕ (if (k? = nil) then cnt?
else ((if (k? = 2) then cnt := cnt? + 1)
⊕ ((k := tail(n)) ⊕(count k))))))
ρ ≡ {0/cnt, [1:2:3:2:5]/p}
(letrec count = (λn.λ_{st}k.(k := head(n))
⊕ (if (k? = nil) then cnt?
else ((if (k? = 2) then cnt := cnt? + 1)
⊕ ((k := tail(n)) ⊕ (count k)))))) [1:2:3:2:5]
ρ ≡ {0/cnt, [1:2:3:2:5]/p}
(letrec count = (k := head([1:2:3:2:5])) ⊕ (if (k? = nil) then cnt?
else ((if (k? = 2) then cnt := cnt? + 1)
⊕ ((k := tail ([1:2:3:2:5])) (count k)))))
ρ ≡ {0/[cnt, [1:2:3:2:5]/p, 1/k}
(letrec count = (if (k? = nil) then cnt?
else ((if (k? = 2) then cnt := cnt? + 1)
⊕ ((k := tail([1:2:3:2:5])) (count k)))))
ρ ≡ {0/cnt, [1:2:3:2:5]/p, 1/k}
(letrec count = ((if (k? = 2) then cnt := cnt? + 1)
⊕ ((k := tail ([1:2:3:2:5])) ⊕ (count k))))
ρ ≡ {0/cnt, [1:2:3:2:5]/p, 1/k}
(letrec count = ((k := tail ([1:2:3:2:5:nil])) ⊕ (count k)))
ρ ≡ {0/cnt, [1:2:3:2:5]/p, [2:3:2:5]/k}
(letrec count = (count k))
ρ ≡ {0/cnt, [1:2:3:2:5]/p, [2:3:2:5]/k}
(letrec count = (λn. λ_{st}k.(k := head(n)) ⊕ (if (k? = nil)
then cnt?
else ((if (k? = 2) then cnt := cnt? + 1)
⊕ ((k := tail(n)) ⊕ (count k))))) [2:3:2:5])
ρ ≡ {0/cnt, [1:2:3:2:5]/p, [2:3:2:5]/k}
...
ρ ≡ {2/cnt, [1:2:3:2:5]/p, nil/k]

4. λ_{st}-계산의 검증

본 절에서는 λ-계산에 상태를 추가한 λ_{st}-계산이 함수형 언어의 기본적인 속성을 그대로 유지함을 보이기 위하여 Church-Rosser 정리를 만족함을 보인다. λ_{st}-계산이 Church-Rosser 정리를 만족함을 보이기 위한 증명 방법의 설정과 본 논문에서 제시되는 정리를 증명하기 위하여 [2]에서 정의된 정의와 기본적인 정리를 활용하며, [2]에서 인용된 정리의 증명은 본 논문에서는 생략한다.

[정의4.1] 강 정규성(SN: strong normalization)
$\forall M \in$에 내해 M에서의 R-감축이 유한하면, R이 강

정규성을 갖는다고 한다. □

[정의4.2] 약 Church-Rosser(WCR : weakly Church-Rosser)
감축 R이 다음을 만족하면 약 Church-Rosser를 만족한다고 한다.
$\forall x, x_1, x_2 [x \rightarrow_R x_1 \wedge \rightarrow_R x_2 \Rightarrow \exists x_3 [x_1 \rightarrow_R x_3 \wedge \rightarrow_R x_3]]$ □

[정의4.3] 교환가능(commute)
r_1, r_2를 감축 의 감축규칙이라고 할 때, 다음을 만족하면 교환가능하다고 한다.
$\forall x, x_1, x_2 [x \rightarrow r_1 x_1 \wedge x \rightarrow r_2 x_2 \Rightarrow \exists x_3 [x_1 \rightarrow r_2 x_3 \wedge x_2 \rightarrow r_1 x_3]]$ □

[정리4.4] Newmann의 정리
감축 R에 대해서 다음이 성립한다.
SN $\wedge$ WCR $\Rightarrow$ Church-Rosser □

[정리4.5] Hindley-Rosen의 정리
R_1, R_2를 임의의 두 감축이라고 할 때, R_1, R_2가 Church-Rosser 정리를 만족하고 $\rightarrow_{R1}$과 $\rightarrow_{R2}$가 교환가능하면, $R_1 R_2$는 Church-Rosser 정리를 만족한다. □

지금까지 도입한 정의와 정리에 의해 λ_{st}-계산이 Church-Rosser 정리를 만족함을 다음과 같이 증명할 수 있다. 먼저, λ_{st}-계산에서 모든 감축규칙에 대한 유한전개(finite development)를 보이고, 상태 관리를 위하여 λ_{st}-계산에서 새로 정의한 $\rightarrow_{st}$ 감축규칙의 강 정규성과 약 Church-Rosser를 보인다. 그리고, $\rightarrow_{\beta\delta}$와 $\rightarrow_{st}$가 교환가능함을 보임으로써 λ_{st}-계산의 모든 감축 $\rightarrow$가 Church-Rosser 정리를 만족함을 증명한다.

4.1 λ_{st}-계산의 강 정규성

순수 λ-계산에서 임의의 항이 종료성을 가지고 있는 경우라면 정규순서계산(normal order evaluation) 방법으로 계산함으로써 반드시 종료될 수 있다. 그러나, 근본적으로 임의의 항이 비종료성을 가지고 있다면 정규순서계산을 이용하더라도 종료될 수 없다. 따라서, λ-항의 종료를 위해서는 비종료성을 지닌 λ-항에 대한 감축은 제외하여야 한다.

마찬가지로, 본 논문에서 제안한 λ_{st}-계산은 순수 λ-계산에 바탕을 두고 있으므로 λ_{st}-계산이 Church-Rosser 정리를 만족함을 보이기 위해서는 비종료성을 지닌 λ_{st}-항에 대한 감축을 제외한 상태에서 증명이 되어야 한다. 이를 위하여 [2]에서 λ-계산을 증명할 때 사용된 방법과 같이 λ_{st}-항에서 감축되어야 할 각 β-감축식(β-redex), δ-감축식(δ-redex)을 표시한 후, 표시된 감축식을 감축해 감으로써 종료됨을 보이도록 한다. 이를 위하여, 〈표 1〉에서 보인 λ_{st}-계산의 각 항에서 감축식을 아래첨자 '0'으로 표시한 모델인 Λ_{st}'을 다음과 같이 정의한다.

[정의4.6] Λ_{st}'의 정의
$M \in \lambda_{st}$-항, x를 한정변수, f를 원시함수라고 할 때, Λ_{st}'을 다음과 같이 정의한다.
(i) $M \in \lambda_{st}$-항 $\Rightarrow$ $M \in \Lambda_{st}'$
(ii) $x, M_1, M_2 \in \Lambda_{st}'$ $\Rightarrow$ $(\lambda_0 x.\ M_1)\ M_2) \in \Lambda_{st}'$
(iii) $f, V \in \Lambda_{st}'$ $\Rightarrow$ $(f_0\ V) \in \Lambda_{st}'$ □

Λ_{st}'에 대한 감축규칙은 $\rightarrow_{st} \cup \rightarrow_{\beta 0} \cup \rightarrow_{\delta 0}$으로 정의된다. 여기서, $\rightarrow_{st}$는 λ_{st}-계산에서 상태를 도입하면서 새로 정의된 감축규칙이며, $\rightarrow_{\beta 0}$, $\rightarrow_{\delta 0}$는 Λ_{st}'에 대한 β-감축식과 δ-감축식을 위하여 [정의4.7]과 같이 정의되는 감축규칙이다.

[정의4.7] Λ_{st}'에 대한 감축규칙 $\rightarrow_{\beta 0}$, $\rightarrow_{\delta 0}$
Λ_{st}'에서 표시된 감축식에 대한 $\rightarrow_{\beta 0}$, $\rightarrow_{\delta 0}$ 감축은 다음과 같이 정의된다.
β_0-감축 : $((\lambda_0 x.\ M)\ N) \rightarrow_{\beta 0} [N/x]M$
δ_0-감축 : $(f_0\ V) \rightarrow_{\delta 0} \delta(f\ V)$ □

다음으로, Λ_{st}'에 대한 감축이 유한함을 증명하기 위하여 감축이 진행되는 상황을 표현할 수 있도록 Λ_{st}'의 각 변수와 원시함수에 가중치를 부여한 $\Lambda_{st}'^*$를 [정의4.8]과 같이 정의한다.

[정의4.8] $\Lambda_{st}'^*$의 정의
(i) x를 한정변수, f를 원시함수라고 할 때, 가중치가 부여된 $\Lambda_{st}'^*$을 다음과 같이 정의한다.

$\forall x, f [x^n, f^n \in \Lambda_{st}'^*, n \in N, n > 0]$
$M \in \Lambda_{st}'^* \Rightarrow (\lambda x.\ M) \in \Lambda_{st}'^*$
$M, N \in \Lambda_{st}'^* \Rightarrow (M\ N) \in \Lambda_{st}'^*$

$M, N \in \Lambda_{st}'^{*} \Rightarrow (\lambda_0 x.\ M) \in \Lambda_{st}'^{*}$

(ii)(i)과 같이 정의되는 $\Lambda_{st}'^{*}$는 다음과 같은 두가지 요소로 분류된다.

$\forall M \in \Lambda_{st}'^{*}[M \equiv (M', I)]$, I는 가중치를 나타내며, M'은 M에서 가중치 I를 제거한 $\Lambda_{st}'^{*}$이다.

$\Lambda_{st}'^{*}$에 대한 감축규칙은 가중치를 가진 한정변수와 원시함수를 위하여 $\rightarrow_{\beta 0}$, $\rightarrow_{\delta 0}$가 [정의4.9]와 같이 확장되는 것 외에는 st'에 대한 감축규칙을 그대로 따른다. 즉, $\Lambda_{st}'^{*}$에서의 가중치는 감축전과 후의 상태를 구별하는 요소로 사용된다.

[정의4.9] $\Lambda_{st}'^{*}$에 대한 감축규칙 $\rightarrow_{\beta 0}^{*}$, $\rightarrow_{\delta 0}^{*}$

$\lambda_0 x.\ x^n$, N, f^n, $V \in \Lambda_{st}'^{*}$라고 할 때, $\rightarrow_{\beta 0}^{*}$, $\rightarrow_{\delta 0}^{*}$ 감축은 다음과 같이 정의된다.

β_0^{*}−감축 : $((\lambda_0 x.\ x^n)\ N) \rightarrow_{\beta 0}^{*}\ N$

δ_0^{*}−감축 : $(f^n, V) \rightarrow_{\delta 0}^{*}\ \delta(f, V)$ □

<표 3> $M \in \Lambda_{st}'^{*}$의 $\|M\|$

<Table 3> $\|M\|$ of $M \in \Lambda_{st}'^{*}$

$\|c\|$	= 0	
$\|x^n\|$	= n	; n : 가중치
$\|f^n\|$	= n	; n : 가중치
$\|(\lambda x.E_1)\ E_2\|$	= 1 + $\|E_1\|$	; $x \notin fv\ E_1$
$\|(\lambda x.E_1)\ E_2\|$	= 1 + $\|E_1\|$ + (#bv E_1)*$\|E_2\|$	; $x \in fv\ E_1$
$\|E_1\ E_2\| = \|E_1\|\|E_2\| = \|E_1 \bullet E_2\| = \|E_1\| + \|E_2\|$		
$\|v?\|$	= 1	
$\|v:=A\|$	= 1 + $\|A\|$	
$\|\lambda_{st} v.E\|$	= 1 + $\|E\|$	
$\|st\ A \triangleright \lambda x.E\|$	= 1 + $\|(\lambda x.E)\ A\|$	
$\|app\ A \triangleright \lambda x.E\|$	= 1+(#sv E')+$\|A\|$+$\|(\lambda x.E)\ EVAL(A)\|$	$:E' \equiv \lambda_{st} v.\ \_app\ A$
$\|v? \triangleright \lambda x.E\|$	= 1 +$\|v?\|$ + $\|(\lambda x.E)\ DEREF(v)\|$	
$\|(v_1\ v_2\ \ldots\ v_n)? \triangleright \lambda x_1 \lambda x_2\ \ldots\ \lambda x_n.E\|$	= 2 * n + $\|\lambda x_1 \lambda x_2\ \ldots\ \lambda x_n.E\|$	
$\|DEREF(v)\| = \|EVAL(E)\|$	= 0	

이제, Λ_{st}'의 강 정규성을 보이기 위하여 $M \in \Lambda_{st}'^{*}$에 대한 $\|M\|$을 〈표 3〉과 같이 정의한다. $\|M\|$은 M이 가지는 가중치의 합을 나타내는 것으로서 $M' \in \Lambda_{st}'$이 정규식으로 변환될 때까지 발생되는 일련의 감축회수를 의미한다. 〈표 3〉에서 c는 상수, x는 한정변수, f는 원시함수, v는 상태변수를 나타낸다. 또한, (#bv E)는 E에서의 한정변수 개수를, (#sv E)는 E에서의 상태변수 개수를 의미한다.

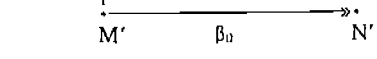
[정의4.10] $(M', \Lambda_{st}'^{*})$의 감소 가중치(decreasing weight)

$M \equiv (M', I) \in \Lambda_{st}'^{*}$라고 할 때, $((\lambda_0 x.P)\ Q)$ M이면 $\forall_x \in P$에 대해서 $\|x\| > \|Q\|$이고, $(f^n\ V) \in M$이면 $\forall f$에 대해 $\|f\| + \|V\| > \|\delta(f, V)\|$일 때, M이 감소 가중치를 갖는다. □

[정리4.11] $(M', I) \in \Lambda_{st}'^{*}$일 때, $\forall M' \in \Lambda_{st}'$에 대해 감소 가중치 I를 부여할 수 있다.

(증명) $\forall M' \in \Lambda_{st}'$ 내에서 발생하는 모든 한정변수와 원시함수에 대해 오른쪽에서 왼쪽으로 $2^n (n \geq 0)$의 가중치를 차례대로 부여하면, $2^n > 2^{n-1} + 2^{n-2} + \ldots + 2^1 + 2^0$가 된다. 따라서, 모든 M'에 감소 가중치를 부여할 수 있다. □

[정리4.12] $M', N' \in \Lambda_{st}'$이고 $M \equiv (M', I) \in \Lambda_{st}'^{*}$일 때, $M' \rightarrow_{\beta 0} N'$이면 $M \rightarrow_{\beta 0}^{*} N$을 만족하는 $(N', I') \in \Lambda_{st}'^{*}$이 존재한다.



[정리4.13] $M \equiv (M', I) \in \Lambda_{st}'^{*}$이고 I가 감소 가중치를 가지며, $M \rightarrow_{st\beta 0}^{*} N$라고 하면, $\forall M, N$에 대하여 $\|M\| > \|N\|$를 만족한다.

(증명) 〈표 3〉에서 정의된 결과를 이용하여 $\Lambda_{st}'^{*}$에 대한 모든 감축규칙이 만족됨을 감축규칙별로 구분하여 보인다.

■ β_0^{*}−감축 : $((\lambda x_0.\ M)\ N) \rightarrow_{\beta 0} [N/x]M$

· $x \notin fv\ M$인 경우

− LHS : $\|((\lambda x_0.\ M)\ N)\| = 1 + \|M\|$

− RHS : $\|[N/x]M\| = \|M\|$ $(\because x \notin fv\ M)$

$\therefore$ LHS > RHS

· $x\ fv\ M$인 경우

− LHS : $\|((\lambda x_0.\ M)\ N)\| = 1 + \|M\| + (\#bv\ M) * \|N\|$

− RHS : $\|[N/x]M\|$

$\therefore$ LHS > RHS

■ δ_0^* −감축 : (fn V)$\rightarrow_{\delta 0}{}^* \delta$(f, V)
− LHS : $\|f^n\ V\| = \|f^n\| + \|V\| = n + \|V\|$
− RHS : $\|\delta(f, V)\|$
∴ LHS 〉 RHS (∵ [정의4.10]과 본 정리의 가정에 의해)

■ λ_{st}v −감축 : (λ_{st}v. M)$\rightarrow_{st}$ M
− LHS : $\|\lambda_{st}v.\ M\| = 1 + \|M\|$
− RHS : $\|M\|$
∴ LHS 〉 RHS

■ := −감축 : (v := M_1) $M_2 \rightarrow_{st} M_2$
− LHS : $\|(v := M_1)\ M2\| = \|v := M_1\| + \|M_2\| = 1 + \|M_1\| + \|M_2\|$
− RHS : $\|M_2\|$
∴ LHS 〉 RHS

■ ‖-감축 :
· ($M_1 \| M_2$)$\rightarrow_{st}$ (λx. M_2) *EVAL*(M_1)인 경우
− LHS : $\|(M_1 \| M_2)\| = \|M_1\| + \|M_2\|$
− RHS : $\|(\lambda x.\ M_2)\ EVAL(M_1)\| = 1 + \|M_2\|$ (∵ x ∉ fv M_2)
∴ LHS 〉 RHS (∵ 1 〈 $\|M_1\|$)
· ($M_1 \| M_2$)$\rightarrow_{st}$ (λx. M_1) *EVAL*(M_2)인 경우 : 전자의 경우와 동일한 결과

■ ⊕ −감축 : ($M_1 \oplus M_2$)$\rightarrow_{st}$ (λx. M_2) *EVAL*(M_1)
− LHS : $\|M_1 \oplus M_2\| = \|M_1\| + \|M_2\|$
− RHS : $\|(\lambda x.\ M_2)\ EVAL(M_1)\| = 1 + \|M_2\|$ (∵ x ∉ fv M_2)
∴ LHS 〉 RHS (∵ 1 〈 $\|M_1\|$)

■ st ▷-감축 : st A ▷ (λx. M)$\rightarrow_{st}$ (λx. M) A
· x ∈ fv M인 경우
− LHS : $\|st\ A \triangleright (\lambda x.\ M)\| = 1 + \|(\lambda x.\ M)\ A\|$
$= 2 + \|M\| + (\#bv\ M)\|A\|$
− RHS : $\|(\lambda x.\ M)\ A\| = 1 + \|M\| + (\#bv\ M)\|A\|$
∴ LHS 〉 RHS
· x ∉ fv M인 경우
− LHS : $\|st\ A \triangleright (\lambda x.\ M)\| = 1 + \|(\lambda x.\ M)\ A\| = 2 + \|M\|$
− RHS : $\|(\lambda x.\ M)\ A\| = 1 + \|M\|$
∴ LHS 〉 RHS

■ app ▷-감축 : app A ▷ (λx. M)$\rightarrow_{st}$ (λx. M) *EVAL*(A)
아래의 증명 과정에서 인용된 E′은 E′ ≡ (... λ_{st}v. ... app A)와 같은 구조를 갖는 항이다.
· x ∈ fv M인 경우
− LHS : $\|app\ A \triangleright (\lambda x.\ M)\| = 1 + (\#sv\ E') + \|A\| + \|(\lambda x.\ M)\ EVAL(A)\|$
$= 1 + (\#sv\ E') + \|A\| + 1 + \|M\| + (\#bv\ M)\|EVAL(A)\|$
$= 2 + (\#sv\ E') + \|A\| + \|M\|$
− RHS : $\|(\lambda x.\ M)\ EVAL(A)\| = 1 + \|M\| + (\#bv\ M)\|EVAL(A)\| = 1 + \|M\|$
∴ LHS 〉 RHS
· x ∉ fv M인 경우
− LHS : $\|app\ A \triangleright (\lambda x.\ M)\| = 1 + (\#sv\ E') + \|A\| + \|(\lambda x.\ M)\ EVAL(A)\|$
$= 1 + (\#sv\ E') + \|A\| + 1 + \|M\|$
− RHS : $\|(\lambda x.\ M)\ EVAL(A)\| = 1 + \|M\|$
∴ LHS 〉 RHS

■ ? ▷-감축 : v? ▷ (λx. M)$\rightarrow_{st}$ (λx. M) *DEREF*(v)
· x ∈ fv M인 경우
− LHS : $\|v? \triangleright (\lambda y.\ M)\| = 1 + \|v?\| + \|(\lambda x.\ M)\ DEREF(v)\|$
$= 2 + 1 + \|M\| + (\#bv\ M)\|DEREF(v)\| = 3 + \|M\|$
− RHS : $\|(\lambda x.\ M)\ DEREF(v)\| = 1 + \|M\| + (\#bv\ E)\|DEREF(v)\|$
$= 1 + \|M\|$
∴ LHS 〉 RHS
· x ∉ fv M인 경우
− LHS : $\|v? \triangleright (\lambda x.\ M)\| = 1 + \|v?\| + \|(\lambda x.\ M)\ DEREF(v)\|$
$= 2 + 1 + \|M\| = 3 + \|M\|$
− RHS : $\|(\lambda x.\ M)\ DEREF(v)\| = 1 + \|M\|$
∴ LHS 〉 RHS

이와 같은 감축규칙의 경우별 분석에 의해 [정리4.13]을 증명하였다. □

[정리4.14] Λ_{st}′에 대한 감축 $\rightarrow_{st\beta\delta 0}$가 강 정규성을 가진다.

(증명) M′ ∈ Λ_{st}′일 때, σ : M′ ≡ M′$_0 \rightarrow_{st\beta 0\delta 0}$ M′$_1 \rightarrow_{st\beta 0\delta 0}$ ... 라고 하면, [정리4.11]에 의해 M′에 대해 감소 가중치 I가 존재하고, [정리4.12]에 의해 다음과 같은 감축이 존재한다.

σ^* : M ≡ (M′$_0$, I)$\rightarrow_{st\beta 0}{}^*{}_{\delta 0}{}^*$ (M′$_1$, I$_1$)$\rightarrow_{st\beta 0}{}^*{}_{\delta 0}{}^*$ ...

이때, [정리4.13]에 의해 각 가중치 I$_i$는 감소하게 되어 $\|(M'_0, I)\| 〉 \|(M'_1, I_1)\| 〉 \|(M'_2, I_2)\| 〉$ ... 가 된다. 따라서, σ와 σ^*는 유한하므로 $\rightarrow_{st\beta\delta 0}$가 강 정규성을 가진다. □

[따름정리4.15] λ_{st}-계산의 감축 $\rightarrow_{st}$가 강 정규성을 가진다.

(증명) λ_{st}-계산에서 일련의 $\rightarrow_{st}$ 감축은 Λ_{st}'에 대한 $\rightarrow_{st\beta\delta 0}$ 감축의 일부에 해당한다. 그리고, Λ_{st}'에서 $\rightarrow_{st\beta\delta 0}$ 감축은 [정리4.14]에 의해서 강 정규성을 가짐을 알 수 있다. 따라서, λ_{st}-계산의 감축 $\rightarrow_{st}$는 강 정규성을 가진다. □

4.2 λ_{st}-계산에 대한 Church-Rosser

본 절에서는 4.1절에서 증명된 정리를 바탕으로 λ_{st}-계산이 Church-Rosser 정리를 만족함을 보인다. 이를 위하여 λ_{st}-계산의 감축 $\rightarrow_{st}$가 약 Church-Rosser 정리를 만족하고, $\rightarrow_{st}$가 $\rightarrow_{\beta\delta}$와 교환가능함을 보인다.

[정리4.16] λ_{st}-계산의 감축 $\rightarrow_{st}$는 약 Church-Rosser 정리를 만족한다. 즉, $M \rightarrow_{st} M_1$이고 $M \rightarrow_{st} M_2$이면, $M_1 \rightarrow_{st} M_3$이고 $M_2 \rightarrow_{st} M_3$인 M_3가 존재한다.

(증명) $\triangle_1$, $\triangle_2$를 감축식이라고 하고, $r_1: M \rightarrow^{\triangle 1}$, $r_2: M \rightarrow^{\triangle 2} M_2$라고 가정하자. $\triangle_1 \equiv \triangle_2$인 경우는 $M_1 \equiv M_2 \equiv M_3$이므로 약 Church-Rosser 정리를 만족한다. $\triangle_1 \neq \triangle_2$인 경우는 〈표 2〉의 감축규칙에서 $\triangle_1 \subset \triangle_2$ 또는 $\triangle_2 \subset \triangle_1$인 경우만이 존재하는데, $\triangle_2 \subset \triangle_1$인 경우에 대해서만 증명하면 된다. $\triangle_1$은 각 감축규칙의 전체 감축식이며, $\triangle_2$는 전체 감축식에 포함된 부분 감축식이다.

■ $\triangle_1 \equiv \lambda_{st}v.E, \rho$

$M \rightarrow^{\triangle 1}$은 $E, \rho \cup \{\perp/v\}$이며, $\triangle_2$는 $E \rightarrow^{\triangle 2} E'$이므로 $M_2 \equiv \lambda_{st}v.E', \rho$이다. 이때, $M_1 \rightarrow^{\triangle 2} M_3$, $M_2 \rightarrow^{\triangle 1} M_3$를 만족하는 $M_3 \equiv E', \rho \cup \{\perp/v\}$이 존재한다.

■ $\triangle_1 \equiv (v := E_1)\, E_2, \cup \{N/v\}$

$M \rightarrow^{\triangle 1} M_1$은 $E_2, \rho \cup \{EVAL(E_1)/v\}$이며, $\triangle_2$는 $E_2 \rightarrow^{\triangle 2} E_2'$이므로 $M_2 \equiv E_2', \rho \cup \{N/v\}$이다. 이때, $M_1 \rightarrow^{\triangle 2} M_3$, $M_2 \rightarrow^{\triangle 1} M_3$를 만족하는 $M_3 \equiv E_2', \rho \cup \{EVAL(E_1)/v\}$이 존재한다.

■ $\triangle_1 \equiv E_1 \| E_2, \rho$

$M \rightarrow^{\triangle 1} M_1$은 $E_1' \| E_2, \rho$이며, $\triangle_2$는 $E_2 \rightarrow^{\triangle 2} E_2'$이므로 $M_2 \equiv E_1 \| E_2', \rho$이다. 이때 $M_1 \rightarrow^{\triangle 2} M_3$, $M_2 \rightarrow^{\triangle 1} M_3$를 만족하는 $M_3 \equiv E_1' \| E_2', \rho$이 존재한다.

■ $\triangle_1 \equiv \triangle_2$인 경우

v?, $\oplus$, st▷, app▷, v?▷ 등은 $\triangle_1 \equiv \triangle_2$인 경우이므로 $M_1 \equiv M_2 \equiv M_3$이다. □

[정리4.17] $\rightarrow_{st}$는 $\rightarrow_{\beta\delta}$와 교환가능(commute)하다.

(증명) $M \rightarrow_{st} M_1$이고 $M \rightarrow_{\beta\delta} M_2$이면, $M_1 \rightarrow \beta\delta\ M_3$이고 $M_2 \rightarrow_{st} M_3$인 M_3가 존재함을 보이면 된다. λ_{st}-감축규칙의 LHS와 $\beta\delta$-감축규칙의 LHS가 중첩되지 않으므로 $\triangle_{st} \neq \triangle_{\beta\delta}$이다. $\triangle_{\beta\delta} \subset \triangle_{st}$인 경우는 [정리4.16]에서 각각의 경우가 교환가능함이 증명되었으므로 $\triangle_{st} \subset \triangle_{\beta\delta}$인 경우에 대하여 증명한다.

■ $\triangle_{\beta\delta} \equiv (\lambda x.E_1)\, E_2$

이때, $\triangle_{st}$는 $E_2 \rightarrow E_2'$인 경우가 가능하다. $\triangle_{st}$가 $E_2 \rightarrow E_2'$일때 다음과 같이 교환가능하다. $M \rightarrow^{\triangle\beta\delta} M_2$는 $[E_2/x]E_1$이며, $E_2 \rightarrow^{\triangle st} E_2'$이므로 $M_1 \equiv (\lambda x.E_1)\, E_2'$이다. 이때 $M_1 \rightarrow^{\triangle st} M_3$, $M_2 \rightarrow^{\triangle\beta\delta} M_3$를 만족하는 $M_3 \equiv [E_2'/x]E_1$이 존재한다.

■ $\triangle_{\beta\delta} \equiv f\ V$

본 논문에서 정의된 원시함수는 스트릭트성 인자를 가지므로 $M_1 \equiv M_2 \equiv M_3$이다. □

[정리4.18] 감축 $\rightarrow_{st}$는 Church-Rosser 정리를 만족한다.

(증명) $\rightarrow_{st}$는 [따름정리4.15]에 의해 강 정규성을 가지며, [정리4.16]에 의해 약 Church-Rosser를 만족한다. 따라서 [정리4.4]에서 보인 Newmann의 정리에 의해 $\rightarrow_{st}$는 Church-Rosser 정리를 만족한다. □

[정리4.19] 감축 $\rightarrow$는 Church-Rosser 정리를 만족한다.

(증명) $\rightarrow_{\beta\delta}$는 [1]에서 증명된 바와 같이 Church-Rosser 정리를 만족하며, $\rightarrow_{st}$는 [정리4.18]에 의해 Church-Rosser 정리를 만족한다. 또한, [정리4.17]에 의해 두 감축은 교환가능하다. 따라서 [정리4.5]의 Hindley-Rosen 정리에 의해 $\rightarrow$는 Church-Rosser 정리를 만족한다.

5. 결 론

순수 함수형 프로그래밍 언어는 상태를 나타내기 위한 방법이 부자연스럽기 때문에 상태를 갖는 일부 알고리즘의 표현과 입출력 및 동적 자료구조의 기술에 어려움이 따른다. 따라서 순수 함수 언어의 대수적 성질을 위배함이 없이 상태를 표현하기 위한 많은 연구가 진행되었다. 그러나 기존 연구의 결과는 함수형 언어의 참조적 투명성을 침해하거나, 형 시스템이나 감축규칙이 복잡하기 때문에 구현이 어렵다는 문제점이 발생된다.

본 논문에서는 상태를 표현함에 있어서 순수 함수형 언어의 대수적 성질을 침해하지 않으면서 감축규칙을 단순화시킨 함수형 언어의 수행모델 λ_{st}-계산을 제안하고, 제안된 모델이 Church-Rosser정의를 만족함을 증명하였다. 제안된 방법을 통해 구문구조의 표현력을 높일 수 있었으며, 감축규칙을 단순화함으로써 구현이 용이할 것으로 기대된다.

모든 상태의 표현이 λ-계산의 구문과 합성될 수 있는 보다 단순화된 계산모델에 대한 설계와 효율적인 구현 방법 등이 앞으로 진행되어야 할 연구과제이다.

참 고 문 헌

[1] S. Abramsky, "The Lazy Lambda Calculus", Research Topics in Functional Programming, pp. 65-116, Addison Wesley, 1988.

[2] H. P. Barendregt, The Lambda Calculus: Its Syntax and Semantics, Vol. 103 of Studies in Logic and the Foundations of Mathematics, North-Holland, 1984.

[3] P. S. Barth, R. S. Nikhil, and Arvind, "M-Structures: Extending a Parallel, Non-strict, Functional Language with state", Functional Programming Language and Computer Architecture, LNCS 523, pp. 538-568, Springer-Verlag, 1991.

[4] A. Bloss, "Update Analysis and Efficient Implementation of Functional Aggregates", Functional Programming Languages and Computer Architecture, pp. 26-38, 1989.

[5] M. Felleisen, "λ-V-CS: An extended λ-calculus for Scheme", ACM Conf. on Lisp and Functional Programming, pp. 72-85, 1988.

[6] M. Felleisen and R. Hieb, "The revised report on the syntactic theories of sequential control and state", Theoretical Computer Science, Vol. 103, pp. 235-271, 1992.

[7] D. K. Gifford and J. M. Lucassen, "Integrating Functional and Imperative Programming", ACM Conf. on Lisp and Functional Programming, pp. 28-38, 1986.

[8] J. C. Guzman and P. Hudak, "Single-Threaded Polymorphic Lambda Calculus", IEEE Symposium on Logic in Computer Science, pp. 333-343, 1990.

[9] P. H. Hartel, "Performance of Lazy Combinator Graph Reduction", Software-Practice & Experience, Vol. 21, No. 3, pp. 299-329, 1991.

[10] G. Hogen, A. Kindler, and R. Loogen, "Automatic Parallelization of Lazy Functional Programs", ESOP '92, European Symposium on Programming, LNCS 582, pp. 254-268, Springer-Verlag, 1992.

[11] P. Hudak, "Concept, Evolution, and Application of Functional Programming Languages", ACM computing surveys, Vol. 21, No. 3, pp. 359-411, 1989.

[12] M. P. Jones, "The Implementation of the Gofer Functional Programming System", Research Report YALEU/DSC/RR-1030, Yale Univ., 1994.

[13] S. D. Kim and W. H. Yoo, "New Combinators for the Efficient Execution of Functional Languages", Proc. InfoScience '93, pp. 381-385, 1993.

[14] J. Launchbury and S. L. Peyton Jones, "Lazy Functional State Threads", Conf. on Program Language Design and Implementation, ACM SIGPLAN Notices, Vol. 29, No. 6, pp. 24-35, June 1994.

[15] E. Moggi, "Computational lambda-calculus and monads", IEEE Symposium on Logic in Computer Science, pp. 14-23, 1989.

[16] M. Odersky, D. Rabin, and P. Hudak, "Call by Name, Assignment, and the Lambda Calculus", 20th ACM POPL, pp. 43-56, 1993.

[17] J. Passia and K.-P. Lohr, "Fips: A Functional-Imperative Language for Explorative Programming", ACM SIGPLAN Notices, Vol. 28, No. 5, May 1993.

[18] S. L. Peyton Jones and P. Wadler, "Imperative functional programming", 20th ACM POPL, pp. 71-84, 1993.

[19] C. G. Ponder, P. C. McGeer, and A. P-C. Ng,

"Are Applicative Languages Inefficient?", ACM SIGPLAN Notices, Vol. 23, No. 6, pp. 135-139, June 1988.

[20] J. C. Reynolds, "Syntactic control of interference, part 2", Automata, Languages and Programming :16th International Colloquium, LNCS 372, pp. 704-722, Springer-Verlag, 1989.

[21] J. G. Riecke and R. Viswanathan, "Isolating Side Effects in Sequential Languages", 22th ACM POPL, pp. 1-12, 1995.

[22] V. Swarup, U. S. Reddy, and E. Ireland, "Assignments for Applicative Languages", Functional Programming Languages and Computer Architecture, LNCS 523, pp. 192-214, Springer-Verlag, 1991.

[23] P. Wadler, "Comprehending Monads", ACM Conf. on Lisp and Functional Programming, pp. 61-78, 1990.

[24] P. Wadler, "The essence of functional progrmming", 19th ACM POPL, pp. 1-14, 1992.

주 형 석

1985년 인하대학교 전자계산학과(이학사).
1987년 인하대학교 대학원 전자계산학과(이학석사).
1997년 인하대학교 대학원 전자계산학과(공학박사).
1987년~현재 유한전문대학 전자계산과 부교수.
관심분야: 프로그래밍 언어(함수언어, 객체지향언어), 계산모델, 병렬 시스템 등임.

김 홍 읍

1988년 한양대학교 수학과(이학사).
1993년 인하대학교 대학원 전자계산학과(공학석사).
1995년~현재 인하대학교 대학원 전자계산공학과 박사과정 재학중.
관심분야: 프로그래밍 언어(함수언어, 객체지향언어), 병렬 처리 등임.

유 원 희

1975년 서울대학교 공과대학 응용수학과 졸업(이학사)
1978년 서울대학교 대학원 계산학 전공(이학석사)
1985년 서울대학교 대학원 계산학 전공(이학박사)
1979년~현재 인하대학교 공과대학 전자계산공학과 교수
관심분야: 프로그래밍 언어(실시간 프로그래밍 언어, 함수 언어).