論文
2-10-3

상태를 갖는 명령형 함수언어의 설계
(A design of the imperative functional language with state)

주 형 석*
(Hyung-Seok Joo)

요 약

함수언어는 여러 가지 유용한 특징에도 불구하고 상태를 나타내기 위한 효율적인 방법을 제공하지 못하고 있다. 함수언어의 표현력을 높이기 위해서는 함수언어의 의미를 위배함이 없이 상태를 명시적으로 나타내기 위한 방법이 요구된다. 이 연구에서는 순수 함수언어의 성질을 위배함이 없이 상태를 표현하기 위한 명령형 함수언어 λst-계산을 설계하였고 제안된 명령형 함수언어를 감축하기 위한 알고리즘을 구성하였다. λ-계산에 명시적인 상태 구성자를 확장한 λst-계산 모델은 상태의 합성 개념을 도입하고 감축규칙을 간략화함으로써 구문구조의 표현력을 향상시켰다.

ABSTRACT

Despite of various useful features, functional languages do not provide an efficient way of representing states. To improve expressiveness of functional language, it is required a method representing explicit state without violating of functional semantic properties. In this paper, imperative functional language, λst-calculus is designed to represent states without compromising the properties of pure functional languages. And we construct an algorithm to reduce proposed imperative functional language. λst-calculus model which is an extension of the λ-calculus model with explicit state constructor without violating their semantic properties. it improves expressiveness of syntax through a concept of state composition and simplified reduction rules.

1. 서론

기존의 명령형 언어(imperative language)와는 다른 파라다임(paradigm)인 함수언어(functional language)는 Backus의 FP[1]가 발표된 이후 프로그래밍 언어와 계산 모델 분야에서 커다란 관심의 대상이 되었다. 이와 같은 연구의 결실로서 ML[2], Miranda[3], HOPE[4], Haskell[5] 등의 많은 함수언어가 개발되었다. 고계함수(higher order function), 지연평가(lazy evaluation), 참조적 투명성(referential transparency), 자료 추상화(data abstraction), 패턴매칭(pattern matching) 등의 여러가지 유용한 특징을 가지고 있는 함수언어는 수학적 함수에 기반을 둔 적용형 언어(applicative language)로서 λ-계산(λ-calculus)을 기본으로 하고 있으며, 프로그램의 전체적인 구조가 함수간의 호출 관계에 의해 구성되어 있다[6, 7, 8].

함수언어는 참조적 투명성을 보장하기 위하여 명령형 언어에서 사용하는 배정문(assignment statement)을 허용하지 않고 함수호출에 의하여 값을 생성하도록 함으로써 각 상태가 임의적으로 변화되지 못하도록

* 정회원 : 유한대학 전자계산과 교수

논문접수 : 2001. 9. 27.
심사완료 : 2001. 10. 15.

※ 이 연구는 1999년도 유한대학 장기해외연구결과논문임.

하고 있다. 이와 같이 상태를 변경시키기 위한 표현을 허용하지 않음으로써 프로그램이 간결하고 추상적이며, 프로그램의 이해와 검증이 용이하며, 묵시적인 병렬성(implicit parallelism)을 자연스럽게 지원한다. 이러한 이유로 함수언어가 병렬 계산 모델의 언어로서 활발하게 연구되고 있지만 상태를 나타내기 위한 알고리즘의 표현이 부자연스럽고 동적 자료구조의 처리가 어렵다는 문제점이 지적되고 있다[9, 10, 11].

함수언어에 상태를 도입하기 위한 연구의 일환으로 비순수 함수언어(impure functional language)인 Standard ML[2], Scheme[12] 등에서는 상태를 나타내기 위하여 제한된 배정문의 표현을 허용하고 있다. 또한 제어의 개념을 포함시켜서 함수언어에 명령형 언어의 특성을 결합하기 위한 시도[9, 13]들이 진행되어 졌다. 그러나, 상태의 개념을 부분적으로 함수언어와 합성하면서 부작용(side effect)의 발생 가능성을 허용함으로써 참조적 투명성을 보장할 수 없게 되었다. 이는 프로그래밍의 편의성을 향상시킬 수 있다는 장점을 가짐에도 불구하고, 수식의 평가 순서가 특정 순서에 의해 유지되어야 하므로 함수언어의 대수적 성질을 위배하고 있다. 이는 순수 함수언어가 지니는 커다란 특징인 묵시적 병렬성을 저해하게 되어, 함수언어가 병렬형 계산 모델에 효율적으로 활용될 수 있다는 특성에 부정적인 영향을 주게 된다[14, 15].

이와 같이 함수언어는 우아한 의미와 여러 가지 특징에도 불구하고, 상태를 표현하기 위한 방법에 문제점을 보이고 있다. 따라서, 함수언어의 대수적 성질을 침해함이 없이 상태의 개념을 도입하며, 구문 구조와 감축 규칙을 단순화함으로써 구현이 용이해 질 수 있는 방법에 대한 연구가 필요로 되고 있다. 본 연구에서는 순수 함수언어의 참조적 투명성을 보장하면서 상태를 도입하기 위한 명령형 함수언어를 설계하고, 이에 대한 감축 알고리즘을 구성한다. 이를 위해 함수언어의 성질을 침해함이 없이 명시적인 상태 구성자(explicit state constructor)를 λ-계산에 확장한 모델을 설계하여 구문 구조의 표현력을 높이고, 감축규칙을 단순화시킨다.

2. 함수언어의 고찰

함수언어에서 주요 개선점으로 지적되고 있는 것은 상태를 효율적으로 나타나기 위한 방법론과 수행과정에서 제기되는 메모리의 효율성에 대한 문제점으로 대별될 수 있다[16, 17, 18]. 또한 함수언어를 기본 모델로 하여 다른 파라다임의 언어와 합성을 하기 위한 많은 시도들도 진행되고 있다[19]. 본 연구의 목표인 명령형 함수언어의 모델은 순수 함수언어의 대수적 성질을 침해함이 없이 함수언어에 상태를 도입하기 위한 방법을 설계하고 구현을 위한 알고리즘을 구성하기 위한 것이다. 상태를 효율적으로 나타내기 위한 본 연구와 관련된 연구 동향은 다음과 같다.

가. 비순수 함수언어에서의 연구

비순수 함수언어인 Standard ML, Scheme 등에서는 수행 순서를 제어하기 위한 제어 연산(control operation)의 개념을 도입하여 제한된 배정문의 사용을 가능하게 하고 있다. 그러나 이들 언어에서는 프로그래밍의 편의성을 향상시키기 위하여 상태의 표현을 허용하였으나 그로 인해 발생될 수 있는 부작용의 발생 가능성을 배제하지 않고 있다. 그 결과 프로그래밍의 편의성은 향상시킬 수 있었으나, 부작용의 발생으로 인하여 참조적 투명성이 파괴될 수 있으므로 프로그램이 항상 특정 순서에 의해 평가되어야 한다는 제한을 가지게 된다.

나. 참조적 투명성을 보장하기 위한 순수 함수언어 이전의 연구

기존 명령형 언어에서 부작용의 발생으로 인해 야기되는 문제를 해결하기 위한 연구는 함수언어 이전의 명령형 언어에서도 Algol 60을 시점으로 하여 많은 연구가 진행되어 졌다. 명령형 언어 측면에서 부작용을 다루기 위한 대표적인 연구로서 Reynolds의 Forsythe[20]를 비롯하여, 비순수 함수언어인 ML이나 Scheme으로 작성된 프로그램의 검증(reasoning)을 위한 Felleisen, Hieb 등의 연구[10], 부작용과 별명(aliasing)의 개념을 형 시스템을 통해 처리하기 위한 Gifford의 Effect 시스템 등이 제안되었다. 그러나 이와 같은 연구들은 순수 함수언어에 상태를 도입하기 위한 시도는 아니었으며, 기존의 λ-계산 관점에서 순수하게 확장된 모델이 되지는 못하였다[14, 15].

다. 순수 함수언어에서의 연구

순수 함수언어에서의 연구방법으로는 상태를 명시적으로 표현하는 방법, 컴파일 기법을 통한 방법,

monad를 이용한 방법 등이 있다. 부작용의 발생을 초래하지 않으면서 순수 함수언어에서 상태를 표현하기 위한 전통적인 방법으로는 상태를 명시적으로 표현하는 방법을 들 수 있다. 그러나, 이 방법에서는 동적 자료의 처리에 있어서 비효율적인 특성을 보이고 있다. 즉, 공유된 동적 자료를 변경하였을 경우 해당 동적 자료를 공유하고 있는 모든 곳에 전체 자료구조의 복사본을 생성하여 명시적으로 전달해야 하는 문제점이 따른다. 상태를 명시적으로 표현하는 문제점을 해결하기 위해서 명시적으로 표현된 상태를 효율적으로 처리하기 위한 컴파일 기법과 언어 개념들이 제안되었으나 컴파일 기법에 의해 얻어지는 정보는 제한적이며 형 시스템이 복잡해지는 문제점을 가지고 있다.

Moggi에 의해 제안된 monad를 이용한 방법[21, 22]은 상태 인자(state parameter)를 명시적으로 표현해야 하는 오버헤드는 없앴으나 상태 정의가 집중화된다는 문제점과 상태를 파괴적으로 갱신하기 위해서는 평가 순서의 명시적인 제어가 필요하다는 문제점을 가지고 있다. 아울러 이러한 접근 방법들은 상태를 값으로 취급하지 못하고 프로그래밍에서 상태를 나타내기 위한 표현이 배제되어 있으며, 동적 자료의 처리가 어렵다는 점이 지적되고 있다[14].

다른 접근법으로서 Swarup의 ILC(Imperative Lambda Calculus)와 Odersky의 λvar 계산 모델을 들 수가 있다. ILC에서는 상태 관련 처리를 추상화한 형태의 연산 observer에 의해 상태를 관리하고 값을 얻는 계산 모델로서 모든 프로그램은 반드시 정규형을 갖는다는 강정규성을 만족하고 있다. 또한 동적 자료의 공유가 가능하고 함수언어의 측면과 명령형 언어의 측면 모두에서 사용될 수 있다는 장점을 가지고 있다. 그러나, ILC에서는 재귀적 개념과 정의된 상태의 값을 직접 참조하는 연산이 배제되어 있으며, 3-단계의 형 시스템(type system)에 의해 상태의 표현이 제한을 받게 된다는 문제점을 보이고 있다. ILC는 상태와 관련된 프로그램의 구성이 제어-전달 형태(continuation- passing style)로 형성되어야만 하는 제약을 가지고 있어서 프로그래밍에 부담을 주고 있다.

λ_{var} 계산 모델에서는 상태 관리 연산자(state transformer)를 이용하여 임의의 상태로부터 값-상태의 짝을 얻는다. λ_{var} 계산 모델은 λ-계산의 성질을 그대로 보존하면서 상태를 표현하고 관리하기 위한 것으로써 함수언어의 개념을 최대한 따르도록 연구된 계산 모델로 평가되고 있다. 그러나, λ_{var} 계산 모델이 함수언어의 개념에 충실하기 위하여 저장(store)의 개념이 없는 배정문을 사용함으로써 구현과정에서 프로그램의 구조를 동적으로 재구축해야 하는 문제점과 감축규칙이 복잡하다는 것 등은 개선되야 할 대표적인 사항들이다.

3. 명령형 함수언어의 설계

본 연구에서는 함수언어의 대수적 성질을 위배함이 없이 함수언어에 상태를 표현하기 위한 명령형 언어의 사양을 확장한 명령형 함수언어를 설계하였다. 함수언어는 λ-계산을 기반으로 하고 있는 언어이므로 상태를 나타내기 위한 언어의 설계를 λ-계산을 확장한 형태로 나타내었다.

λ_{st}-계산은 순수 함수언어의 확장 λ-계산(enriched λ-calculus)에 상태를 표현하기 위한 상태 구성자를 확장한 계산 모델이다. 프로그래밍의 표현력과 편의성을 향상하기 위하여 함수언어에서 상태를 표현하고 조작할 수 있도록 정의한 λ_{st}-계산의 구문구조는 <표 1>과 같다.

(1) 순수 λ-항

λ_{st}-계산은 순수 λ-계산을 확장한 모델이므로 기본적인 λ-계산의 항들을 갖는다.

<표 1> λ_{st}-계산의 구문구조

<Table 1> Syntax definition of λst-calculus

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

(2) 상태변수의 정의

λ_{st}-항 내에서 상태를 표현하기 위한 상태변수(mutable variable) 정의와 영역의 설정은 λ_{st}v.E에 의해 이루어진다. 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.

(3) 상태 관리 연산자

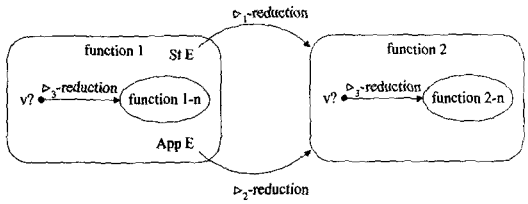
상태 관리 연산자(state transformer) ?, :=는 스트릭트성(strict) 인자를 가지는 연산자이다. v?은 현재의 상태에 영향을 주지 않고 상태변수 v의 값을 참조하기 위한 상태 관리 연산자로서 ?이 상태 합성 연산자 ▷과 결합되면 참조된 값을 ▷에 의해 합성될 함수의 인자로 전달한다. v:=A는 식 A의 평가 결과를 상태변수 v에 저장하는 것으로서 :=의 좌변은 상태변수이며 우변은 λ_{st}-항으로 구성된다.

상태 관리 연산자 st와 app는 상태변수의 영역을 확장하기 위한 것으로서 st는 상태 합성 연산자 ▷과 결합되어 현재의 함수 내에서 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 평가하고 app I가 포함된 함수 내에서 정의된 모든 상태를 파괴한 후 I의 정규형을 합성될 함수의 인자로 전달

(4) 상태 합성 연산자

상태 합성 연산자는 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 연구에서 정의한 λ_{st}-계산에서의 상태 관리 연산자와 상태 합성 연산자의 관계에 의해 발생되는 상태식의 합성 개념을 표현한 것이다.



[그림 1] 상태식의 합성

[Fig. 1] Composition of expression with state

[그림 1]에서 $\triangleright_1$, $\triangleright_2$로 표현된 상태 합성 연산자는 함수와 함수사이에서 발생하는 함수 적용으로서 함수1 내의 임의의 상태식을 함수2의 인자로 합성하여 β-감축을 유도한다는 공통점을 가지고 있다. 반면, $\triangleright_1$, $\triangleright_2$의 차이점은 다음과 같은 두 가지로 정의된다. $\triangleright_1$은 함수1 내의 상태 관리 연산자 st에 의해 생성되는 상태식을 함수2의 인자로 전달하면서 함수1 내에서 정의된 현재의 모든 상태까지 그대로 전달시켜 주는 반면에, $\triangleright_2$는 함수1 내의 상태 관리 연산자 app에 의해 생성되는 상태식을 함수2의 인자로 합성한 후 함수1에서 정의된 모든 상태를 파괴한다. 그리고, st $\triangleright_1$ 상태식 합성에서 st는 비스트릭트성(non-strict) 인자를 취하지만 app $\triangleright_2$ 상태식 합성에서의 app는 스트릭트성 인자를 취하게 된다.

(5) 확장 원시 연산자

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 병렬 연산자 ‖, 순차연산자 ⊕의 이항 원시 연산자를 정의한다. 확장 원시 연산자 ‖, ⊕의 구성 예와 감축의 의미를 나타내면 다음과 같다.

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

EVAL(E)는 E를 평가한 결과를 의미하며, λx.E에서 x는 E에서 자유변수가 아니다. ‖는 피연산자1과 피연산자2 사이에 수행순서 관계를 고려할 필요가 없는 경우를 표현하기 위한 연산자이다. 그리고, ⊕는 피연산자1을 먼저 수행한 후 피연산자2를 수행해야 할 수식을 표현하기 위한 것으로서 피연산자1에 의한 상태 변화를 피연산자2에서 활용하는 경우에 사용된다.

4. 명령형 함수언어의 감축기 설계

4.1 감축규칙과 감축 알고리즘

λ_{st}-항을 평가하는데 사용되는 λ_{st}-계산의 감축규칙은 <표 2>와 같이 정의한다. <표 2>에서 EVAL(E)는 λ_{st}-항 E를 평가한 결과를, fv(E)는 λ_{st}-항 E에서 자유변수의 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.

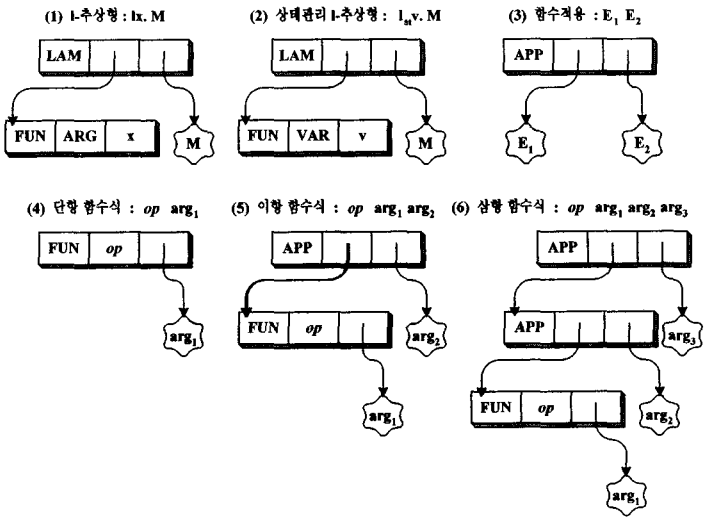
상태를 갖는 λ_{st}-계산의 코드는 순수 λ-계산에 상태와 관련된 구문이 추가된 프로그램으로서 λ_{st}-계산 프로그램에 대한 그래프 생성기에 의해 힙 메모리에 이진 그래프 형태로 구성된다. 그리고, 그래프 감축(graph reduction)은 힙 메모리에 구성된 λ_{st}-계산 프로그램 그래프의 루트노드에 대한 포인터를 그래프 생성기로부터 전달받아 앞에서 제안된 감축규칙에 따라 감축하면서 결과값을 생성하게 된다. 이때, 그래프 감축기가 감축을 진행하기 위해서는 λ_{st}-계산 프로그램 그래프 상에서 감축대상이 되는 감축식은 루트노드부터 깊이우선순서로 탐색함으로써 가능하며, 탐색과정에서 탐색해온 경로는 스파인 스택이 저장된다.

λ_{st}-계산 프로그램에서의 기본이 되는 단위 프로그램 그래프의 구성은 [그림 2]와 같다. 그래프를 구성하는 각 노드의 형태는 3개의 정보 필드를 갖는 노드로 구성한다. 각 노드의 첫 번째 정보는 태그로서 LAM(lambda 추상화), APP(함수적용), FUN(함수) 등을 나타내기 위한 태그이며 각각의 태그에 대한 인자로서 두 개의 나머지 정보가 채워진다. 두 번째 필드의 정보는 포인터 또는 함수를 나타내는데 함수는 일반함수와 타입함수가 될 수 있다. 타입함수는 const(상수), arg(한정변수), var(상태변수)를 갖는다. 세 번째 필드의 정보는 포인터 또는 상수, 한정변수, 상태변수의 값을 갖는다. [알고리즘 1]은 <표 2>의 감축규칙을 바탕으로 λ_{st}-계산 프로그램을 감축하기 위한 그래프 감축 알고리즘이다.

<표 2> λ_{st}-계산의 감축규칙

<Table 2> Reduction rule for λst-calculus

· β-감축	: $(\lambda x.E_1)E_2$	→	$E_1[E_2/x]$
· δ-감축	: f V	→	δ(f, V)
· 확장 감축			
λ_{st}	: λ_{st}v. E ρ	→	E ρ∪{⊥/v}
?	: v? ρ∪{N/v}	→	N ρ∪{N/v}
:=	: (v := E_1) E_2 ρ∪{N/v}	→	E_2 ρ∪{*EVAL*(E_1)/v}
‖	: E_1 ‖ E_2 ρ	→	⎧ ($\lambda x.E_2$) *EVAL*(E_1) ρ ;x∉*fv*(E_2) ⎩ ($\lambda x.E_1$) *EVAL*(E_2) ρ ;x∉*fv*(E_1)
⊕	: E_1 ⊕ E_2 ρ	→	($\lambda x.E_2$) *EVAL*(E_1) ρ ;x∉*fv*(E_2)
st▷	: st A ▷ (λy. E) ρ	→	(λy. E) A ρ
app▷	: app A ▷ (λy. E) ρ	→	(λy. E) *EVAL*(A) { }
? ▷	: v? ▷ (λy. E) ρ∪{N/v}	→	(λy. E) N ρ∪{N/v}



[그림 2] λ_{st}-계산 프로그램에서의 기본구조

[Fig. 2] Basic structure in λ_{st}-calculus program graph

[알고리즘 1] λ_{st}-calculus 프로그램 그래프 감축 알고리즘

```
Input    : λst-calculus 프로그램 그래프의 root node 주소,
spine stack 포인터
Output   : 상수 값
Method   :
   Evaluate(node_address, spine_stack)
   { IP ← node_address;
   do {
      while(IP.tag is 'APP' node)
      { push IP onto spine stack;
      IP ← IP.left;
      }

      if (IP.tag is 'LAM' node)
          switch(IP.left->left)
          { case 'ARG':
          substitute IP.left->right with subgraph pointed by
          spine_stack-1;
          pop two element from spine stack;
          IP ← IP.right;
      case 'VAR':
          IP ← (spine_stack-1)->right;
          pop two element from spine stack;
          }
      else    /* IP.tag is 'FUN' node */
          switch(IP.left)
          { case '?':
          return the value of (IP->right)->right;
          pop one element from spine stack;
      case ':=':
          Evaluate((spine_stack-1)->right, spine_stack);
          substitute the state with return value on
          (IP->right)->right;
          IP ← (spine_stack-2)->right;
          pop three element from spine stack;
      case '‖':
          Evaluate(IP->right, spine_stack)
          Evaluate((spine_stack-1)->right, spine_stack);
          pop two element from spine stack;
      case '⊕':
          Evaluate(IP->right, spine_stack)
          Evaluate((spine_stack-1)->right, spine_stack);
          pop two element from spine stack;
      case '▷':
          if ( (IP->right)->left is 'st' )
          { allocate a node;
            IP ← address of allocated node;
          IP->tag ← 'APP';
          IP->left ← (spine_stack-1)->right;
          IP->right ← (spine_stack->right)->right;
          pop two element from spine_stack;
          }
          else if ( (IP->right)->left is 'app' )
          { allocate a node;
            IP ← address of allocated node;
          IP->tag ← 'APP';
          IP->left ← (spine_stack-1)->right;
          IP->right ← Evaluate((spine_stack->right)->right,
          spine_stack);
          clear the states;
          pop two element from spine_stack;
          }
          else
          { allocate a node;
            IP ← address of allocated node;
          IP->tag ← 'APP';
          IP->left ← (spine_stack-1)->right;
          IP->right ← Evaluate(spine_stack->right,
          spine_stack);
          pop two element from spine_stack;
          }

   } while(IP is not ROOT_NODE);
   /* ROOT_NODE: root node address of program graph */
   }
```

4.2 적용 예

λ_{st}-계산을 이용해서 누적합 생성 프로그램에 전달되는 인자 값들의 누적 결과를 생성하는 프로그램의 예이다. 이를 위한 누적합 생성 프로그램은 상태변수의 정의를 포함하여 다음과 같이 작성할 수 있으며, 각 단계에서 밑줄로 표현된 식이 감축되어질 식이다.

((accumulate1) 0)▷(λctr. ctr 1▷λx. ctr 2)	ρ≡{}
→ ((λinit. λ_{st}cnt. (cnt:=init) ⊕ (st λinc. (cnt:=cnt?+inc) ⊕ (cnt?))) 0) ▷(λctr. ctr 1▷λx. ctr 2)	ρ≡{}
→ (λ_{st}cnt. (cnt:=0) ⊕ (st λinc. (cnt:=cnt?+inc) ⊕ (cnt?))) ▷(λctr. ctr 1▷λx. ctr 2)	ρ≡{}
→ ((cnt:=0) ⊕ (st λinc. (cnt:=cnt?+inc) ⊕ (cnt?))) ▷(λctr. ctr 1▷λx. ctr 2)	ρ≡{⊥/cnt}
→ (st λinc. (cnt:=cnt?+inc) ⊕ (cnt?))▷(λctr. ctr 1▷λx. ctr 2)	ρ≡{0/cnt}
→ ((λctr. ctr 1▷λx. ctr 2) (λinc. (cnt:=cnt?+inc) ⊕ (cnt?)))	ρ≡{0/cnt}
→ ((λinc. (cnt:=cnt?+inc) ⊕ (cnt?)) 1)▷(λx. (λinc. (cnt:=cnt?+inc) ⊕ (cnt?)) 2)	ρ≡{0/cnt}
→ ((cnt:=cnt?+1) ⊕ (cnt?))▷(λx. (λinc. (cnt:=cnt?+inc) ⊕ (cnt?)) 2)	ρ≡{0/cnt}
→ (cnt?)▷(λx. (λinc. (cnt:=cnt?+inc) ⊕ (cnt?)) 2)	ρ≡{1/cnt}
→ ((λx. (λinc. (cnt:=cnt?+inc) ⊕ (cnt?)) 2) 1)	ρ≡{1/cnt}
→ ((λinc. (cnt:=cnt?+inc) ⊕ (cnt?)) 2)	ρ≡{1/cnt}
→ ((cnt:=cnt?+2) ⊕ (cnt?))	ρ≡{1/cnt}
→ cnt?	ρ≡{3/cnt}
→ 3	ρ≡{3/cnt}

5. 결론

본 연구에서는 순수 함수언어의 대수적 성질을 위배함이 없이 상태를 도입한 명령형 함수언어를 설계하고 이를 구현하기 위한 감축규칙과 감축 알고리즘을 보였다. 상태를 갖는 명령형 순수 함수언어 λ_{st}-계산은 λ-계산에 상태를 표현하기 위한 상태 구성자를 도입한 확장된 λ-계산 모델이다. λ_{st}-계산에서 상태의 합성 개념을 도입하고 구문구조를 간략화함으로써 프로그래밍의 표현력과 편의성을 향상하였다.

앞으로 추가적으로 진행될 연구형태로서는 제안된 계산모델의 실험적 수행모델을 구현하기 위한 전처리기와 감축기의 구현이 필요하다. 또한 현재 구성된 모델을 기본모델로 다른 파라다임의 언어와 합성하는 방안에 대한 연구를 통한 멀티파라다임 언어의 설계가 앞으로의 하나의 연구과제로 제기된다.

※ 참고문헌

[1] Backus, J. B., 1978, "Can Programming Be Liberated from the von Nuemann Style? A Functional Style and Its Algebra of Program", CACM, Vol. 21, No. 8, pp. 613-642.

[2] Milner, R., 1984, "A Proposal for Standard ML", ACM Conf. on Lisp and Functional Programming, pp. 184-197.

[3] Turner, D. A., 1985, "Miranda : A Non-strict Functional Programming Language with Polymophic Types", LNCS 201, SpringerVerlag, pp. 1-16.

[4] Field, A. J. and Harrison, P. G., 1988, *Functional Programming*, Addison Wesley.

[5] Hudak, P., Peyton Jones, S. L., and Wadler, P., 1992, "Report on the Functional Programming Language Haskell, version 1.2", ACM

SIGPLAN Notices, Vol. 27, No. 5.

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

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

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

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

[10] Felleisen, M. and Hieb, R., 1992, "The Revised Report on the Syntactic Theories of Sequential Control and State", Theoretical Computer Science, Vol. 103, pp. 235-271.

[11] Joo, H. S., 1997, "An Improvement of Expressiveness and Performance in Functional Languages with State", Ph.D. thesis, Inha Univ..

[12] Rees, J. and Clinger, W.(eds.), 1986, " The Revised3 Report on the Algorithmic Language Scheme", ACM SIGPLAN Notices, Vol. 21, No. 12, pp. 37-79.

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

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

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

[16] Stefanovic, D., McKinley, K. S., and Moss, J. E., 1999, "Age-Based Garbage Collection", ACM SIGPLAN Conf. on OOPSLA, pp. 370-381.

[17] Mountjoy, J., 2000, "The Spineless Tagless G-machine, naturally", Proc. of the ACM SIGPLAN International Conf. on Functional Programming, pp. 163-173.

[18] Domani, T., Kolodner, E. K., and Petrank, E., 2000, "A Generational On-the-fly Garbage Collectior for Java", ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 274-284.

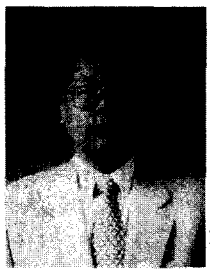
[19] McNamara, B. and Smaragdakis, Y., 1999, "Functional Programming in C++", Proc. of the ACM SIGPLAN International Conf. on Functional Programming, ACM SIGPLAN Notices, Vol. 34, No. 1, pp. 118-129.

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

[21] Moggi, E., 1989, "Computational Lambda-calculus and Monads", IEEE Symposium on Logic in Computer Science, pp. 14-23.

[22] Wadler, P., 1992, "The Essence of Functional Progrmming", 19th ACM Symposium on POPL, pp. 1-14.

주 형 석



1985년 인하대학교
전자계산과 졸업
1987년 인하대학교
전자계산과 이학석사
1997년 인하대학교
전자계산공학과 공학박사
2000년 2월 ~2001년 1월
SFSU 객원연구원
1987년 ~ 현재
유한대학 전자계산과 교수