다단계 프로그램에서 프로그램 생성 단계의 자유변수 제거

(Closed-code-conversion: Transforming Open Code Multi-staged Programs into Closed Ones)

어 현 준 † 이 광 근 ††
(Hyunjun Eo) (Kwangkeun Yi)

요 약 다단계 프로그래밍이란 매크로 프로그래밍, 부분 계산(partial evaluation), 실행시간 코드 생성(runtime code generation)등을 포섭하는 일반적인 방법론으로, 계산을 여러 단계로 나누어 각 단계에 주어진 부분 입력을 통해 다음 단계를 전문화(specialize)함으로써 효율적인 계산을 수행하게 해 준다. 다단계 프로그램은 일반적인 계산 외에 코드(다음 단계의 프로그램)를 생성, 조립 및 실행시킬 수 있다.

본 논문은 코드에 자유변수를 허용하는 다단계 프로그램을 코드에 자유변수가 없는 다단계 프로그램으로 변환하는 방법을 제안한다. 코드에 존재하는 자유변수는 동적으로 바인딩(binding)되기 때문에 이를 구현하기 어려운 문제가 있다. 자유변수가 있는 코드는 환경을 입력으로 받는 함수의 코드로 변환하고, 필요한 환경은 코드를 조립하는 시점에 넘겨 줌으로써 코드에서 자유변수를 제거할 수 있다. 이렇게 자유변수가 제거된 다단계 프로그램은 Davies와 Pfenning이 제안한 방법에 의해 단계가 없는 람다 계산(lambda-calculus)으로 변환된 후 람다 계산법에 의해 실행되어질 수 있다.

키워드 : 다단계 프로그램, 자유변수가 있는 코드, 자유변수 없애기

Abstract We present a transformation which converts open-code multi-staged programs into closed ones. Staged computation, which explicitly divides a computation into separate stages, is a unifying framework for existing program generation systems. Because a multi-staged program generates another program, which can also generate a third program and on, the implementation of a multi-staged language is not straightforward. Dynamic binding of (lexically free) variables in code also makes the implementation of a multi-staged language hard. By converting each code into code of function which takes environment for free variables as its argument and giving an actual environment at the code-composition site, we can transform a open-code program into a closed-code one. Combining with Davies and Pfenning's method, our closed-code-conversion enables the implementation of the unstaged language to be useful for executing multi-staged programs. We also prove the correctness of our conversion: the converted program is equivalent to the original program, and the converted program does not have open code.

Key words : Multi-staged program, open code, closed-code-conversion

· 이 논문은 2007 한국컴퓨터종합학술대회에서 '코드에 자유변수가 있는 다단계 프로그램에서 자유변수 없애기'의 제목으로 발표된 논문을 확장한 것임

† 정 회 원 : 포항공과대학교 미래기술사업단 교수
hyunjuneo@postech.ac.kr
†† 종신회원 : 서울대학교 컴퓨터공학부 교수
kwang@cse.snu.ac.kr

논문접수 : 2007년 9월 28일
심사완료 : 2009년 1월 2일

1. 서 론

다단계 프로그래밍은 계산을 여러 단계로 나누어 현재 단계에서는 다음 단계의 계산에 필요한 코드를 생성하고 마지막 단계에 전체적인 계산을 수행하도록 하는 프로그래밍 방법이다. 다단계 프로그래밍은 매크로 프로그래밍[1,2], 부분 계산(partial evaluation)[3,4], 실행시간 코드 생성 기법(run-time code generation)[5,6] 등을 포섭하는 일반적인 방법론으로 주로 코드의 재사용, 부분 입력에 대한 전문화(specialization) 등에 사용된다.

다단계 프로그래밍에서 계산은 크게 두 가지로 구분

이 된다. 그 중 하나는 일반적인 계산을 수행하는 부분이고, 나머지 하나는 다음 단계의 계산에 필요한 코드[1]를 생성하는 부분이다. 즉 3단계 프로그래밍을 통해 계산을 한다면 0단계는 1단계 코드를, 1단계는 2단계 코드를 생성하고, 마지막 2단계 코드를 실행함으로써 모든 계산이 완료된다. 이 과정에서 각 단계에 부분적으로 주어지는 입력들을 이용하여 다음 단계의 계산을 전문화(specialize)하여 단계를 나누지 않은 프로그램보다 더 효율적인 계산을 할 수 있도록 해 준다.

다단계 프로그래밍을 안전하게 하기 위해서, 다단계 프로그래밍 언어에 대한 타입 이론이 활발하게 연구되어져 왔다. 그 중 대표적인 것이 양상 논리(modal logic)를 이용한 타입 시스템이다. 가장 먼저, 1996년에 Davies와 Pfenning이 코드에 자유변수를 허용하지 않는 다단계 언어에 대한 타입 시스템을 제안하였다[7]. 그 후 코드에 자유변수를 허용하도록 다단계 언어의 타입 시스템을 만드는 연구가 10여년 동안 진행되어 오다가 2006년 Kim, Yi와 Calcagno에 의해 리습(Lisp)의 매크로를 완벽하게 지원하면서 자유변수를 허용하는 다단계 언어의 타입 시스템이 제안 되었다[5].

코드에 자유변수를 허용하는 타입 시스템을 이용한 프로그램은 자유변수를 허용하지 않는 프로그램에 비해 더 효율적인 코드를 생성할 수 있다. 코드에 자유변수를 허용하지 않으면 모든 코드 부품들이 닫힌(closed) 형태가 되어야 하므로 모든 자유변수를 입력으로 받는 함수의 형태로 만들어 주어야 한다. 따라서, 프로그램을 복잡하게 만들어 프로그래밍이 어려워지며, 이를 실행하는 데 있어서 자유변수를 허용하는 프로그램보다 비효율적일 수 밖에 없다.

Kim, Yi와 Calcagno에 의해 자유변수를 허용하는 다단계 언어와 그 타입 시스템이 제안되었지만[8], 이를 어떻게 구현할 수 있을 지에 대한 연구는 아직까지 진행되어진 바가 없다. 다단계 언어를 구현하는 방법으로는 다단계 언어를 실행시킬 수 있는 가상 기계를 고안하는 방법과 다단계 프로그램을 단계가 없는 보통의 프로그램으로 변환한 다음 그 프로그램을 실행시키는 방법 등이 있다.

본 논문은 자유변수를 허용하는 다단계 언어를 단계가 없는 보통의 프로그램을 변환하여 실행시키는 연구의 일 부분으로 시작되었다. 자유변수를 허용하지 않는 다단계 언어를 단계가 없는 보통의 프로그램으로 변환하는 방법은 Davies와 Pfenning에 의해 간략하게 제시된 바가 있다. 따라서, 본 논문에서 제안하는 자유변수를 허용하는 다단계 언어를 자유변수를 허용하지 않는 다단계 언어로 변환하는 과정과 Davies와 Pfenning의 방법을 결합하여 자유변수를 허용하는 다단계 프로그램을 단계가 없는 보통의 프로그램으로 변환할 수 있다.

본 논문에서 제시하는 방법은 자유변수가 있는 함수를 자유변수가 없는 함수로 바꾸어 주는 방법인 클로져 변환(closure conversion)[9,10]과 매우 유사하다. 클로져(closure)란 함수와 그 함수가 정의될 때의 환경(environment)을 쌍(pair)으로 묶은 것을 말한다. 클로져 변환을 통해 함수는 환경을 레코드(record)의 형태로 입력 받도록 변환되고, 모든 자유변수는 입력으로 받은 환경의 필드(field)가 되어 함수 내에 자유변수가 더 이상 존재하지 않게 된다.

자유변수가 있는 다단계 프로그램을 자유변수가 없도록 변환할 때 가장 큰 문제점은 자유변수가 동적으로 바인딩(binding)되는 것이다. 즉, 코드에 있는 자유변수는 코드가 정의될 때의 환경에 영향을 받는 것이 아니라 코드가 사용될 때의 환경에 영향을 받는 것이다. 따라서, 본 논문에서 제안하는 방법은 클로져 변환[9,10]에서처럼 코드가 생성될 때의 환경을 코드에 넘겨주는 것이 아니라 코드가 사용될 때의 환경을 코드에 넘겨주도록 변환한다.

본 논문은 다음과 같이 구성된다. 2절에서는 다단계 언어를 소개하고, 3절과 4절에 각각 코드에 자유변수를 허용하지 않는 타입 시스템과 허용하는 타입 시스템을 정의한다. 5절에서는 코드에 자유변수가 있는 프로그램을 자유변수가 없는 프로그램으로 변환하는 방법을 제시하고, 제시된 변환 방법이 올바르게 작동함을 증명한다. 즉, 변환된 프로그램과 변환되기 전 프로그램이 같은 의미를 가지고, 변환된 프로그램은 코드에 자유변수가 없음을 증명한다.

2. 다단계 언어

2.1 다단계 언어의 정의

다단계 언어의 핵심 문법은 다음과 같다.

$$e ::= c \mid x \mid \lambda x.e \mid e_1 e_2 \\ \mid \mathbf{box}\, e \mid \mathbf{unbox}_k\, e \mid \mathbf{eval}\, e$$

다단계 언어는 일반적인 람다 계산(lambda calculus)외에 코드를 생성(**box**), 코드 부품들을 치환하여 조합(**unbox**), 생성된 코드를 실행(**eval**)시킬 수 있는 방법을 제공한다. **box, unbox, eval**은 각각 리습(Lisp)의 매크로 생성(`), 치환(,) 및, 실행(**eval**)에 대응된다.

다단계 언어의 실행 의미는 그림 1과 같이 정의된다. 실행 규칙 $e \xrightarrow{n} v$의 의미는 프로그램 e가 n단계

1) 코드란 실행되는 프로그램이 아닌, 생성되는 프로그램을 의미한다. 즉, 1단계 이상의 프로그램을 코드라 한다.

에서 실행되어 끝난다면 그 값은 v라는 의미이다. 다단계 언어의 값은 각 단계별로 다른 형태를 가지며 다음과 같이 정의된다.

$$\begin{aligned} v^n \in Val^n ::= {} & c \mid \lambda x.e \mid \mathbf{box}\, v^1 && \text{if } n=0 \\ ::= {} & c \mid x \mid \lambda x.v^n \mid v^n v^n \\ & \mid \mathbf{box}\, v^{n+1} \mid \mathbf{eval}\, v^n \mid \mathbf{unbox}_k\, v^{n-k} && \text{if } n > k \geq 0 \end{aligned}$$

0 단계에서는 일반적인 계산이 실행되거나 코드를 생성 또는 실행시킬 수 있다. 즉 함수 호출, 코드 생성(**box**) 또는 코드 실행(**eval**)이 0 단계에서 실행되어 진다. 함수 호출은 일반적인 람다 계산법과 같이 함수의 형식 인자(formal argument) x를 실 인자(actual argument) v로 치환하여 함수의 몸체(body)를 실행하게 된다. 다단계 언어에서의 치환 규칙은 그림 2와 같이 정의 되어진다.

1단계 이상에서는 일반적인 계산(람다 계산에서는 함수 호출)은 실행되지 않고 단지 코드를 생성하거나 조합하는 일만 할 수 있다. 즉 n단계를 실행 중 **box**를 만나면 단계가 하나 증가하여 n+1단계가 되고, $\mathbf{unbox}_k$를 만나면 단계가 k만큼 감소하여 n-k단계가 된다. 코드를 끼워 넣은 $\mathbf{unbox}_k$연산은 현재의 단계와 $\mathbf{unbox}_k$를 통해 내려가는 단계가 일치하여 0 단계까지 떨어졌을 때만 수행된다. $\mathbf{unbox}_k$에서 k는 1단계에서 생성된 코드를 임의의 k단계에서 재사용 할 수 있게 해 준다.

$$\frac{}{c \xrightarrow{n} c} \qquad \frac{}{x \xrightarrow{n} x}(n>0)$$

$$\frac{}{\lambda x.e \xrightarrow{0} \lambda x.e} \qquad \frac{e \xrightarrow{n} v}{\lambda x.e \xrightarrow{n} \lambda x.v}(n>0)$$

$$\frac{e_1 \xrightarrow{0} \lambda x.e_3 \quad e_2 \xrightarrow{0} v \quad [v/x^0]^0 e_3 \xrightarrow{0} v}{e_1\ e_2 \xrightarrow{0} v}$$

$$\frac{e_1 \xrightarrow{n} v_1 \quad e_2 \xrightarrow{n} v_2}{e_1\ e_2 \xrightarrow{n} v_1\ v_2}$$

$$\frac{e \xrightarrow{n+1} v}{\mathbf{box}\, e \xrightarrow{n} \mathbf{box}\, v}$$

$$\frac{e \xrightarrow{0} \mathbf{box}\, v}{\mathbf{unbox}_n\, e \xrightarrow{n} v} \qquad \frac{e \xrightarrow{n-k} v}{\mathbf{unbox}_k\, e \xrightarrow{n} \mathbf{unbox}_k\, v}(n>k)$$

$$\frac{e \xrightarrow{0} \mathbf{box}\, v^1 \quad v^1 \xrightarrow{0} v}{\mathbf{eval}\, e \xrightarrow{0} v} \qquad \frac{e \xrightarrow{n} v}{\mathbf{eval}\, e \xrightarrow{n} \mathbf{eval}\, v}(n>0)$$

그림 1 다단계 언어의 실행 의미

$$\begin{aligned} [v/x^m]^n c &= c \\ [v/x^m]^n y &= v, && \text{if } x=y \text{ and } n=m \\ &= y, && \text{otherwise} \\ [v/x^m]^n(\lambda y.e) &= \lambda y.e, && \text{if } x=y \text{ and } n=m \\ &= \lambda y.([v/x^m]^n e), && \text{otherwise} \\ [v/x^m]^n(e_1\ e_2) &= ([v/x^m]^n e_1)\ ([v]^n e_2) \\ [v/x^m]^n(\mathbf{box}\, e) &= \mathbf{box}\, ([v/x^m]^{n+1} e) \\ [v/x^m]^n(\mathbf{unbox}_k\, e) &= \mathbf{unbox}_k\, ([v/x^m]^{n-k} e) \\ [v/x^m]^n(\mathbf{eval}\, e) &= \mathbf{eval}\, ([v/x^m]^n e) \end{aligned}$$

그림 2 치환 규칙: m단계 변수 x를 n단계에서 v로 치환

2.2 다단계 프로그래밍 예

다단계 프로그래밍은 주로 프로그램의 입력이 여러 개가 있을 때 그 프로그램의 부분 입력에 특화된 프로그램을 생성하여 나머지 입력이 주어졌을 때 모든 입력에 대해 한번에 계산하는 것 보다 효율적인 프로그램을 생성하고자 할 때 사용된다.

다음과 같은 **x**의 **n**-제곱승을 구하는 **power**함수를 생각해 보자.

```
fun power x n =
   if n=0 then 1 else x * (power x (n-1))
```

만약 이 함수가 **x**의 3-제곱승을 구하는 데 자주 사용되어진다면 다음과 같은 **power3** 함수를 사용하여 필요없는 재귀 호출과 조건문의 실행을 줄일 수 있을 것이다.

```
fun power3 x = x*x*x
```

다단계 프로그램을 통해 이와 같이 주어진 함수로부터 부분 입력에 대해 특화된 함수를 생성할 수 있다. 다음의 **spower**와 **fastpower**함수는 **n**에 특화된 **power** 함수를 생성한다. 이 예에서는 프로그램의 가독성을 높이기 위해서 **box**와 $\mathbf{unbox}_1$ 대신 리습(Lisp)의 콰지-쿼트(quasi-quote) 기호를 사용하였다.

```
fun spower n =
if n=0 then `1
else `(x *,(spower (n-1)))
fun fastpower n = eval `(λx.,(spower n))
```

spower 함수는 **fastpower** 함수의 몸체에 해당하는 코드를 만든다. **spower 1**을 실행하면 '**(x *,(spower 0))**라는 코드가 만들어 지고, **(spower 0)** 자리에 **spower 0**의 결과인 '**1**이 치환되어 들어가 '**(x*1)**이라는 코드를 만들게 된다. 이와 같이 **spower** 함수의 결과는 **x**라는 자유변수를 생성하게 되고, 이 자유변수는 **spower** 함수를 호출하는 **fastpower**에서 바인딩이 이루어진다. 즉, 코드의 자유변수는 코드가 생성될 때의 환경에 영향을 받는 것이 아니라 코드가 사용될 때의 환경에 영향을 받는 동적 바인딩을 따른다.

fastpower 함수는 **spower** 함수의 결과에 자유변수 **x**를 바인딩 시켜 함수의 형태로 만들어 준다. 따라서, **fastpower 3**를 실행하면 **power3**와 같이 3-제곱승을 구하는 함수 **λx.x*x*x*1**이 생성되어지고, **fastpower 5**를 실행하면 5-제곱승을 구하는 함수 **λx.x*x*x*x*x*1**이 생성되어 진다.

3. 코드에 자유변수를 허용하지 않는 타입 시스템

코드에 자유변수를 허용하지 않는 타입 시스템은 Davies와 Pfenning에 의해 제안되었다[2]. 이 타입 시

스템은 S4 양상 논리(modal logic)와 커리-하워드 대응(Curry-Howard correspondence)관계를 가진다.

타입 B는 상수 타입 ι, 함수 타입 $B \to B$, 또는 코드 타입 $\Box B$로 이루어진다. 코드 타입 $\Box B$는 양상 논리의 필연성(necessity)에 대응 되며, B는 현재 단계가 아닌 다음 단계에서 수행되어질 코드이며, 다음 단계의 코드가 그 이후의 임의의 단계에서 재사용 될 수 있음을 의미한다.

$$\begin{aligned} B \in Type &::= \iota \mid B \to B \mid \Box B \\ \Delta \in TypeEnvironment &= Var \xrightarrow{\text{fin}} Type \end{aligned}$$

타입 규칙은 그림 3과 같다. 타입 판단식 $\Delta_0 \cdots \Delta_n \vdash e : B$는 타입 환경(type environment) $\Delta_0 \cdots \Delta_n$하에서 표현식(expression) e가 타입 B를 가짐을 의미한다. 타입 환경 $\Delta_0 \cdots \Delta_n$는 각각 0 단계부터 n단계까지 변수들에 대한 타입 정보를 가진다. 코드를 생성(**box**)할 때는 자유변수를 허용하지 않기 때문에 항상 빈(empty) 환경으로부터 시작하고, n-k(k>0)단계에서 코드 타입이었던 표현식은 임의의 n단계에서 재사용되어질 수 있다.

$$\frac{}{\Delta_0 \cdots \Delta_n \vdash_c c : \iota}$$

$$\frac{\Delta_n(x) = B}{\Delta_0 \cdots \Delta_n \vdash_c x : B}$$

$$\frac{\Delta_0 \cdots \Delta_n + x : B_1 \vdash_c e : B_2}{\Delta_0 \cdots \Delta_n \vdash_c \lambda x.e : B_1 \to B_2}$$

$$\frac{\Delta_0 \cdots \Delta_n \vdash_c e_1 : B_1 \to B \quad \Delta_0 \cdots \Delta_n \vdash_c e_2 : B_1}{\Delta_0 \cdots \Delta_n \vdash_c e_1\ e_2 : B}$$

$$\frac{\Delta_0 \cdots \Delta_n \varnothing \vdash_c e : B}{\Delta_0 \cdots \Delta_n \vdash_c \mathbf{box}\ e : \Box B}$$

$$\frac{\Delta_0 \cdots \Delta_{n-k} \vdash_c e : \Box B}{\Delta_0 \cdots \Delta_n \vdash_c \mathbf{unbox}_k\ e : B}$$

$$\frac{\Delta_0 \cdots \Delta_n \vdash_c e : \Box B}{\Delta_0 \cdots \Delta_n \vdash_c \mathbf{eval}\ e : B}$$

그림 3 코드에 자유변수를 허용하지 않는 타입 규칙

4. 코드에 자유변수를 허용하는 타입 시스템

코드에 자유변수를 허용하는 타입 시스템은 Kim, Yi와 Calcagno에 의해 제안되었다[8]. 이 타입 시스템이 자유변수를 허용하지 않는 타입 시스템과 다른 점은 코드 타입에 자유 변수들의 환경을 명시해 주는 것이다. 따라서 코드 타입은 $\Box(\Gamma \rhd A)$의 형태를 가지며 A타입의 코드 타입이며 필요한 자유 변수의 환경은 Γ라는 의미이다.

$$\begin{aligned} A \in Type &::= \iota \mid A \to A \mid \Box(\Gamma \rhd A) \\ \Gamma \in TypeEnvironment &= Var \xrightarrow{\text{fin}} Type \end{aligned}$$

타입 규칙은 그림 4와 같이 정의되며, 코드에 자유변수를 허용하지 않는 타입 시스템의 규칙과 거의 유사하다. 코드를 생성(**box**)할 때는 자유변수의 환경을 타입에 기록할 수 있으므로, 그 환경을 이용하여 코드의 타입을 줄 수 있다. 코드를 조합(**unbox**)할 때는 그 때의 환경과 코드에 기록된 자유변수의 환경이 일치할 때만 그 코드를 사용할 수 있다. 코드가 자유변수 x를 필요로 하는데 사용되는 곳에서 x에 대한 환경이 없다면 이는 잘못된 코드를 조합하게 되기 때문이다. 생성된 코드를 실행(**eval**)하기 위해서는 그 코드의 환경이 빈(empty) 환경일 때만 실행할 수 있다. 즉, 실행될 코드가 자유변수가 없는 코드일 때에만 실행할 수 있다. 코드를 실행한다는 것은 1단계의 코드를0단계로 끌어 내려서 실행한다는 의미인데, 자유변수가 있는 코드를 실행하게 되면 서로 다른 단계의 두 변수(1단계의 자유변수와 0단계의 환경에 있는 변수)가 서로 연동될 수 있어 프로그램의 의미를 파악하기 힘들어지기 때문이다.

$$\frac{}{\Gamma_0 \cdots \Gamma_n \vdash_o c : \iota}$$

$$\frac{\Gamma_n(x) = A}{\Gamma_0 \cdots \Gamma_n \vdash_o x : A}$$

$$\frac{\Gamma_0 \cdots \Gamma_n + x : A_1 \vdash_o e : A_2}{\Gamma_0 \cdots \Gamma_n \vdash_o \lambda x.e : A_1 \to A_2}$$

$$\frac{\Gamma_0 \cdots \Gamma_n \vdash_o e_1 : A_1 \to A \quad \Gamma_0 \cdots \Gamma_n \vdash_o e_2 : A_1}{\Gamma_0 \cdots \Gamma_n \vdash_o e_1\ e_2 : A}$$

$$\frac{\Gamma_0 \cdots \Gamma_n \Gamma \vdash_o e : A}{\Gamma_0 \cdots \Gamma_n \vdash_o \mathbf{box}\ e : \Box(\Gamma \rhd A)}$$

$$\frac{\Gamma_0 \cdots \Gamma_{n-k} \vdash_o e : \Box(\Gamma_n \rhd A)}{\Gamma_0 \cdots \Gamma_n \vdash_o \mathbf{unbox}_k\ e : A}$$

$$\frac{\Gamma_0 \cdots \Gamma_n \vdash_o e : \Box(\varnothing \rhd A)}{\Gamma_0 \cdots \Gamma_n \vdash_o \mathbf{eval}\ e : A}$$

그림 4 코드에 자유변수를 허용하는 타입 규칙

5. 코드에 자유변수가 있는 프로그램을 자유변수가 없는 프로그램으로 변환

이 절에서는 코드에 자유변수가 있는 다단계 프로그램을 코드에 자유변수가 없는 다단계 프로그램으로 변환하는 방법을 제안한다. 코드는 사실 함수로 생각할 수 있기 때문에[7], 코드에서 자유변수 없애기는

함수에서 자유변수 없애기인 클로져 변환(closure conversion)과 유사한 방법을 사용할 수 있다. 즉, 표현식 e가 e'으로 변환된다면 **box** e는 환경을 레코드(record) 형태로 입력으로 받는 함수의 코드 **box** ($\lambda env.e'$)으로 변환하면 된다. 이때 코드 내에 있는 자유변수 x는 환경 레코드 env의 x필드(field)를 접근하도록 변환된다.

코드에 자유변수가 있는 다단계 프로그램에서 자유변수들은 동적으로 바인딩(binding)된다. 즉, 코드가 생성(**box**)될 때의 환경에 영향을 받는 것이 아니라 코드가 사용(**unbox**)될 때의 환경에 영향을 받는 것이다. 따라서, 코드가 사용될 때의 환경을 코드에 넘겨주도록 변환한다. 코드가 사용될 때의 환경은 레코드(record)의 형태로 변환되어 함수에 전달된다.

5.1 자유변수가 없는 다단계 레코드 언어

2절에서 정의한 다단계 언어는 레코드를 지원하지 않기 때문에 레코드를 지원하도록 다단계 언어를 확장해야 한다. 레코드를 지원하기 위해서는 레코드를 생성하고 레코드 필드를 접근하는 방법을 제공해야 한다.

레코드는 다음의 형태로 정의된다.

$$\{l_i = e_i\}^{(m)}$$

그 의미는 레코드에 m개의 서로 다른 필드가 있고, 그 중 i $(0 \le i \le m)$번째 필드의 이름은 l_i이며 그 필드의 값은 e_i를 계산한 값이 된다. 레코드 필드의 접근은 $e.l$의 형태로 정의되며 e가 l필드를 가지는 레코드일 때 그 필드의 값이 계산된 결과가 된다.

$$\frac{e_i \xrightarrow{n} v_i}{\{l_i = e_i\}^{(m)} \xrightarrow{n} \{l_i = v_i\}^{(m)}} \qquad \frac{e \xrightarrow{n} \{l_i = e_i\}^{(m)}}{e.l_j \xrightarrow{n} v_j}$$

코드에 자유변수가 없도록 변환된 프로그램은 레코드를 가지므로, 자유변수를 허용하지 않는 타입 시스템은 다음과 같이 레코드 타입을 포함해야 한다.

$$B \in Type \qquad ::= \iota \mid B \to B \mid \Box B \mid \{l_i : B_i\}^{(m)}$$
$$\Delta \in TypeEnvironment = Var \xrightarrow{\text{fin}} Type$$

레코드 표현식과 마찬가지로 레코드 타입 $\{l_i{:}B_i\}^{(m)}$은 m개의 서로 다른 필드를 가지는 레코드에 대한 타입으로 각 필드 l_i의 타입은 B_i임을 의미한다. 타입 규칙은 일반적인 레코드 타입 규칙과 동일하게 정의된다.

$$\frac{\Delta_0 \cdots \Delta_n \vdash_c e_i : B_i}{\Delta_0 \cdots \Delta_n \vdash_c \{l_i : e_i\}^{(m)} : \{l_i : B_i\}^{(m)}}$$

$$\frac{\Delta_0 \cdots \Delta_n \vdash_c e : \{l_j : B_j\}^{(m)}}{\Delta_0 \cdots \Delta_n \vdash_c e.l_i : B_i}$$

5.2 변환 규칙

자유변수를 허용하는 프로그램에서 코드 e가 실행되는 환경은 종속변수(bound variable)와 자유변수(free variable)를 포함하고 있다. 즉 프로그램 텍스트 상에서 어떤 코드 단계의 변수가 자유변수일 지라도 코드 단계에서의 변수는 동적으로 바인딩(binding)되어야만 실행될 수 있기 때문에 정적 의미 구조인 타입 시스템에서도 타입 환경에 종속변수와 자유변수에 대한 타입을 모두 가지고 있어야 한다.

따라서, 그림 5에 정의된 자유변수가 없는 코드로 변환하는 규칙 $B_0 \cdots B_n \; ; \; F_0 \cdots F_n \vdash e \Rightarrow e'$은 현재 환경에서 종속변수의 집합이 $B_0 \cdots B_n$, 자유변수의 집합이 $F_0 \cdots F_n$일 때 표현식 e를 e'으로 변환해 준다. 각 B_i와 F_i는 서로 소이고, 위에서도 언급한 바와 같이 B_i와 F_i의 합집합은 타입환경 Γ의 정의역(domain)과 같다.

$$\frac{}{B_0 \cdots B_n; \mathcal{F}_0 \cdots \mathcal{F}_n \vdash c \Rightarrow c}$$

$$\frac{x \in \mathcal{B}_n}{\mathcal{B}_0 \cdots \mathcal{B}_n; \mathcal{F}_0 \cdots \mathcal{F}_n \vdash x \Rightarrow x} \qquad \frac{x \in \mathcal{F}_n}{\mathcal{B}_0 \cdots \mathcal{B}_n; \mathcal{F}_0 \cdots \mathcal{F}_n \vdash x \Rightarrow env.x}$$

$$\frac{\mathcal{B}_0 \cdots \mathcal{B}_{n-1}(\mathcal{B}_n \cup \{x\}); \mathcal{F}_0 \cdots \mathcal{F}_{n-1}(\mathcal{F}_n \setminus \{x\}) \vdash e \Rightarrow e'}{\mathcal{B}_0 \cdots \mathcal{B}_n; \mathcal{F}_0 \cdots \mathcal{F}_n \vdash \lambda x.e \Rightarrow \lambda x.e'}$$

$$\frac{\mathcal{B}_0 \cdots \mathcal{B}_n; \mathcal{F}_0 \cdots \mathcal{F}_n \vdash e_1 \Rightarrow e_1' \quad \mathcal{B}_0 \cdots \mathcal{B}_n; \mathcal{F}_0 \cdots \mathcal{F}_n \vdash e_2 \Rightarrow e_2'}{\mathcal{B}_0 \cdots \mathcal{B}_n; \mathcal{F}_0 \cdots \mathcal{F}_n \vdash e_1\ e_2 \Rightarrow e_1'\ e_2'}$$

$$\frac{\mathcal{B}_0 \cdots \mathcal{B}_n\emptyset; \mathcal{F}_0 \cdots \mathcal{F}_n\mathcal{F}_{n+1} \vdash e \Rightarrow e'}{\mathcal{B}_0 \cdots \mathcal{B}_n; \mathcal{F}_0 \cdots \mathcal{F}_n \vdash \mathbf{box}\ e \Rightarrow \mathbf{box}\ (\lambda env.e')} \quad \begin{array}{l}\text{where } e\text{'s type is } \Box(\Gamma \rhd A) \\ \text{and } \mathcal{F}_{n+1} = dom(\Gamma)\end{array}$$

$$\frac{\mathcal{B}_0 \cdots \mathcal{B}_{n-k}; \mathcal{F}_0 \cdots \mathcal{F}_{n-k} \vdash e \Rightarrow e'}{\mathcal{B}_0 \cdots \mathcal{B}_n; \mathcal{F}_0 \cdots \mathcal{F}_n \vdash \mathbf{unbox}_k\ e \Rightarrow (\mathbf{unbox}_k\ e')\ \mathrm{Rcd}(\mathcal{B}_n, \mathcal{F}_n)}$$

$$\frac{\mathcal{B}_0 \cdots \mathcal{B}_n; \mathcal{F}_0 \cdots \mathcal{F}_n \vdash e \Rightarrow e'}{\mathcal{B}_0 \cdots \mathcal{B}_n; \mathcal{F}_0 \cdots \mathcal{F}_n \vdash \mathbf{eval}\ e \Rightarrow (\mathbf{eval}\ e')\ \{\}}$$

$$\mathrm{Rcd}(\mathcal{B}_n, \mathcal{F}_n) = \{x = x \mid x \in \mathcal{B}_n\} \cup \{x = env.x \mid x \in \mathcal{F}_n\}$$

그림 5 자유변수 제거 변환

각각의 표현식에 대한 변환 방법은 다음과 같다.

- 상수 c의 경우에는 변환할 필요 없이 그대로 둔다.
- 변수 x의 경우, 현재 환경에서 x가 종속변수이면 그대로 두고, 자유변수이면 입력으로 주어진 환경 레코드 env의 x필드를 참조하도록 변환한다.
- 함수 $\lambda x.e$의 경우, 함수의 몸체 e안에서 x는 종속변수이므로, e를 변환할 때는 x를 종속변수의 집합에 넣고 자유변수의 집합에서 빼준 후 변환한다. 몸체 e가 e'으로 변환이 된다면 전체 함수는 $\lambda x.e'$으로 변환이 된다.
- 함수 호출 e_1 e_2의 경우, e_1과 e_2가 현재 환경에서 각각 e_1'과 e_2'으로 변환이 된다면 전체 함수 호출식은 e_1' e_2'으로 변환된다.
- 코드 생성 **box** e의 경우, e가 e'으로 변환된다면 자유변수를 입력으로 받는 함수 $\lambda env.e'$의 코드인 **box** ($\lambda env.e'$)으로 변환된다. 코드 부분인 e를 변환

할 때, 새로운 코드가 시작되므로 종속 변수는 없고 자유변수만 존재하게 된다. 주의할 점은 자유변수의 집합은 텍스트 상에 드러난 유효범위(lexical scope)에 의해 결정되는 것이 아니라 실제 실행되는 상황에 의해 결정된다는 것이다. 예를 들어 (λy. **box** (**unbox**$_1$ y)) (**box** x)를 실행하면 **box** x라는 결과가 나오기 때문에, 람다 함수 안에 있는 코드의 자유변수집합은 공집합이 아니라 $\{x\}$가 된다. 이렇게 동적으로 결정되는 자유변수 집합은 이미 타입시스템을 통해서 코드 타입에 기록되어 있으므로, 타입 정보를 이용해서 쉽게 자유변수 집합을 찾을 수 있다. 예를 들어 위 예에서 람다 함수 안의 코드의 타입은 $\square(\emptyset \triangleright A)$가 아니라 $\square(\{x{:}A\} \triangleright A)$이다.

- 코드 치환 **unbox**$_k$ e의 경우, e가 e'으로 변환된다면 (**unbox**$_k$ e') Rcd(B_n,F_n)으로 변환한다. e'이 환경을 입력으로 받는 함수의 코드 타입이기 때문에 **unbox**$_k$를 하게 되면 환경을 입력으로 받는 함수가 나오고 이 함수에 현재의 환경을 레코드로 만든 Rcd(B_n,F_n)을 넘겨 주면 된다. 이때 종속 변수 x에 대한 필드는 x=x와 같이 그대로 넘겨주면 되지만, 자유변수 y는 변환에 의해 $env.y$로 이미 변환된 상태이므로 y=$env.y$의 형태로 넘겨주어야 한다. 예를 들어, 다음 프로그램은

(λy. **box**(**unbox**$_1$ y)) (**box** x)

다음과 같이 변환된다.

(λy. **box**(λenv. (**unbox**$_1$ y) {x=$env.x$}))
(**box** (λenv.($env.x$)))

- 코드 실행 **eval** e의 경우, e가 e'으로 변환된다면 (**eval** e') {}으로 변환한다. **eval**은 단계를 변화시키지 않고 바인딩도 일어나지 않으므로 자유변수 집합, 종속변수 집합도 역시 변하지 않는다. **eval**을 실행한 결과도 역시 환경을 입력으로 받는 함수가 나오고, 자유변수를 허용하는 타입 시스템에서도 코드를 실행하기 위해서는 그 코드가 항상 자유변수가 없어야 하기 때문에 환경은 {}로 넘겨주면 된다.

5.3 변환의 정확성(correctness)

본 논문에서 제안한 변환이 정확함을 다음의 두 가지 정리를 통해 증명하였다.

정리 1은 변환하기 전과 변환하고 난 후의 프로그램이 동치 관계(equivalence relation)에 있는 결과를 계산함을 보여준다. 변환된 프로그램의 동치 관계는 그림 6과 같이 정의된다. 그림 6의 동치 관계는 환경을 입력으로 받는 부분들만 부분 실행한 결과가 정확히 일치함을 의미한다.

정리 1. *자유변수가 있는 프로그램 e가 자유변수가 없는 e'으로 변환되고, e를 실행한 결과가 v, v를 변환한 결과가 v'이라면 변환된 프로그램 e'은 v'과 동치 관계에 있는 v''을 결과로 낸다.*

$$
\begin{array}{rl}
c \equiv c & \\
x \equiv x & \\
\lambda x.v \equiv \lambda x.v' & \text{iff } v \equiv v' \\
(\lambda env.v_1)\ v_2 \equiv v' & \text{iff } [v_2/env^0]^0 v_1 \equiv v' \\
v_1\ v_2 \equiv v_1'\ v_2' & \text{iff } v_1 \neq \lambda env.v,\ v_1 \equiv v_1',\ \text{and } v_2 \equiv v_2' \\
\textbf{box}\ v \equiv \textbf{box}\ v & \text{iff } v \equiv v' \\
\textbf{unbox}_k\ v \equiv \textbf{unbox}_k\ v' & \text{iff } v \equiv v' \\
\textbf{eval}\ v \equiv \textbf{eval}\ v' & \text{iff } v \equiv v' \\
\{x_i = v_i\}^{(m)} \equiv \{x_i = v_i'\}^{(m)} & \text{iff } v_i \equiv v_i' \\
v.x \equiv v' & \text{iff } v = \{x = v', \cdots\}
\end{array}
$$

그림 6 변환된 프로그램의 동치(equivalence) 관계

증명. 그림 1에서 정의한 실행의미 트리의 크기에 따른 귀납법으로 증명 가능하다. 증명에서 가장 어려운 부분은 코드 치환식에 관한 부분이다. 치환식 **unbox** e를 변환하면 (**unbox** e') Rcd(B_n,F_n)이라는 함수 호출식이 되는데, 1단계 이상의 코드 또는 함수 내에서는 함수 호출이 실제로 일어나지 않기 때문에 이를 그림 6에서 정의한 동치 관계에 의해 함수 호출이 일어나는 효과(환경 변수 env를 실제 환경으로 치환)를 가지도록 하는 것이 증명의 핵심이다. 구체적으로, e가 **box** v로 실행될 때, **unbox** e는 (**unbox** e') Rcd(B_n,F_n)이라는 프로그램으로 변환된다. 변환되기 전 프로그램 **unbox** e를 실행시킨 결과 v를 치환할 때의 환경 B_n과 F_n에서 변환한 결과를 v', 치환되기 전 환경 Ø와F_1'에서 변환한 결과를 v''이라 하자. 변환되기 전과 변환된 후 프로그램을 각각 실행하면 v와 ($\lambda env.v''$) Rcd(B_n,F_n)이 된다. 변환된 프로그램의 실행 결과 ($\lambda env.v''$) Rcd(B_n,F_n)은 그림 6의 동치관계에 의해 [Rcd(B_n,F_n)/env^0]$^0v''$와 동치 관계이다:

(1) ($\lambda env.v''$) Rcd(B_n,F_n) ≡ [Rcd(B_n,F_n)/env^0]$^0v''$.

v'을 구할 때의 환경 B_n, F_n과 v''을 구할 때의 환경 Ø, F_1'을 각각 반대로 v''과 v'에 치환하면 서로 부족한 부분을 채워 주어 동치 관계가 된다:

(2) [Rcd(B_n,F_n)/env^0]$^0v''$ ≡ [Rcd(Ø,F_1')/env^0]$^0v'$.

[Rcd(Ø,F_1')/env^0]$^0v'$와 ($\lambda env.v'$) Rcd(Ø,F_1')는 다시 그림 6에 의해 동치 관계이다:

(3) [Rcd(Ø,F_1')/env^0]$^0v'$ ≡ ($\lambda env.v'$) Rcd(Ø,F_1').

자유변수가 없는 Rcd(Ø,F_1')를 환경레코드로 넘겨 주는 것은 결국 아무 정보도 주지 않는 것과 같다:

(4) ($\lambda env.v'$) Rcd(Ø,F_1') ≡ v'.

따라서, (1)~(4)에 의해 변환된 프로그램을 실행한 값 ($\lambda env.v''$) Rcd(B_n,F_n)과 변환되기 전 프로그램을 실행한 값 v를 변환한 값 v'은 서로 동치관계이다:

(5) ($\lambda env.v''$) Rcd(B_n,F_n) ≡ v'.

나머지 부분은 귀납법에 의해 쉽게 증명된다. ■

정리 2는 변환된 프로그램이 더 이상 자유변수를

가지고 있지 않음을 보여준다. 이는 변환된 프로그램이 자유변수를 허용하지 않는 타입시스템을 통과할 수 있음을 보여주면 된다. 그림 7과 같이 자유변수를 허용하는 타입은 자유변수를 허용하지 않는 타입으로 변환되어질 수 있고 이를 이용하여 정리 2를 증명할 수 있다.

$$\|\iota\| = \iota$$
$$\|A_1 \to A_2\| = \|A_1\| \to \|A_2\|$$
$$\|\Box(\Gamma \rhd A)\| = \Box(\|\Gamma\| \to \|A\|)$$

$$\|\{x_i : A_i\}^{(m)}\| = \{x_i : \|A_i\|\}^{(m)}$$

그림 7 타입 변환

정리 2. *자유변수가 있는 프로그램 e가 자유변수가 없는 e'으로 변환되고, e가 자유변수를 허용하는 타입 규칙에 의해 타입이 A임을 증명할 수 있다면 변환된 프로그램 e'은 자유변수를 허용하지 않는 타입 규칙에 의해 타입 A를 변환한 타입 ||A|| 임을 증명할 수 있다.*

증명. 그림 7에서와 같이 자유변수를 허용하는 타입과 자유변수를 허용하지 않는 타입 사이에 대응관계가 있기 때문에 프로그램 e의 구조에 대한 귀납법으로 쉽게 증명된다. ■

6. 코드의 자유변수 제거 예

앞의 2.2절에서 소개한 n-제곱승을 구하는 다단계 프로그램에 대해서 생각해 보자.

```
fun spower n =
  if n=0 then `1
  else `(x * ,(spower (n-1)))
fun fastpower n = eval `(λx. ,(spower n))
```

함수 spower의 결과 중 else문에 의해 생성되는 결과는 x를 자유변수로 가지는 코드이다.

위 프로그램에서 앞서 제시한 방법에 의해 자유변수를 제거하면 다음과 같은 자유변수가 없는 프로그램으로 변환된다.

```
fun spower n =
  if n=0 then `(λe.1)
  else `(λe. e.x *,(spower (n-1)){x=e.x})
fun fastpower n =
  eval `(λe.λx. ,(spower n) {x=x}) {}
```

모든 코드는 환경 레코드 e를 입력으로 받는 함수의 코드로 변환된다. 함수 spower의 else문의 결과 코드에서 자유변수 x는 환경 레코드 e의 x필드를 참조하도록 변환되었기 때문에 코드에서 자유변수는 모두 사라지게 된다. 즉, 자유변수는 환경 레코드의 필드로 변환되게 되고, 필요한 필드들은 코드를 사용하는 시점에 각각 {x=e.x}와 {x=x}로 제공해 줌으로써 자유변수가 제거된다. spower안에서 spower의 결과를 사용하는 ,(spower (n-1))의 경우, x가 자유변수 이므로 넘겨주는 환경 레코드는 {x=e.x}가 된다. 함수 fastpower에서 spower를 사용할 때는 x가 종속 변수 이므로 필요한 환경 레코드는 {x=x}가 된다.

이렇게 코드의 자유변수가 제거된 다단계 프로그램은 Davies와 Pfenning의 방법[7]에 의해 아래와 같이 단계를 구분하지 않는 프로그램으로 변환된다.

```
fun spower n =
  if n=0 then λb.λe.1
  else (λb.λe. e.x *
            (spower (n-1)) () {x=e.x}
       )
fun fastpower n =
  (λb.λe.λx. (spower n) () {x=x}) () {}
```

변환의 핵심은 코드는 람다 함수로, 코드의 사용은 임의의 값(이 예에서는 ())을 코드가 변환된 함수에 적용하는 것으로 바꿔주면 된다.

단계가 제거된 프로그램은 다단계 프로그램에서와 같이 단계를 명시하고 있지는 않지만 내부적으로는 여전히 단계를 가지고 있다. 따라서, 다단계 프로그래밍 언어에 대한 컴파일러(compiler)나 실행기(interpreter)를 구현하지 않고도 프로그램을 단계적으로 실행할 수 있게 된다.

즉, 다음의 네 단계를 거쳐 다단계 프로그램을 단계가 없는 일반적인 컴파일러나 실행기로 구현할 수 있다.

1. 코드에 자유변수가 있는 다단계 프로그램의 작성.
2. 자유변수가 제거된 다단계 프로그램으로 변환.
3. 단계가 없는 일반 프로그램으로 변환.
4. 변환된 프로그램을 실행

7. 결 론

본 논문은 코드에 자유변수를 허용하는 다단계 프로그램을 코드에 자유변수가 없는 다단계 프로그램으로 변환하는 방법을 제안하였다. 제안된 방법은 코드를 환경을 입력 받는 함수의 코드로 변환하고, 필요한 환경은 코드가 사용되는 시점에서 제공함으로써 자유변수가 없는 코드로 변환해 준다. 본 논문에서 제시된 방법을 통해 자유변수가 없게 변환된 다단계 프로그램은 Davies와 Pfenning이 제안한 방법에 의해 단계가 없는 람다 계산(lambda-calculus)으로 변환되어져 람다 계산법에 의해 실행되어질 수 있다. 제안된 변환이 정확함을 증명하였고, 이를 구현하여 실제로 사용할 수 있도록 하는 것이 차후 과제이다.

참 고 문 헌

[1] Guy L. Steele. *Common Lisp the Language, 2nd*

edition. Digital Press, 1990.
[2] Paul Graham. *On Lisp: an advanced technique for Common Lisp*. Prentice Hall, 1994.
[3] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic generation*. Prentice Hall, 1993.
[4] Oliver Danvy. Type-directed partial evaluation. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 242-257, ACM, Jan 1996.
[5] M. Leone and Peter Lee. Optimizing ML with run-time code generation. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 137-148. ACM, Jun 1996.
[6] Massimilian Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kasshoek. C and tcc: a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21:324-369, Mar 1999.
[7] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 258-270, ACM, Jan 1996.
[8] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for Lisp-like multi-staged languages. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 257-268, ACM, Jan 2006.
[9] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 271-283, ACM, Jan 1996.
[10] Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48-86, Jan 1997.

어 현 준

1996년 한국과학기술원 학사. 1998년 한국과학기술원 석사. 2006년 한국과학기술원 박사. 2006년~2008년 서울대학교 연구원. 2008년~현재 포항공과대학교 연구조교수

이 광 근

1987년 서울대학교 학사. 1990년 Univ. of Illinois at Urbana-Champaign 석사 1993년 Univ. of Illinois at Urbana-Champaign 박사. 1993년~1995년 Bell Labs, Murray Hill 연구원. 1995년~2003년 한국과학기술원 교수. 2003년~현재 서울대학교 교수