# 다단계 프로그램을 위한 타입 기반의 실행흐름 분석

진민식 ○ 이광근

서울대학교 컴퓨터공학부 프로그래밍 연구실

{msjin○,kwang}@ropas.snu.ac.kr

# Type based Control Flow Analysis of Multi-Staged Program

Minsik Jin○ Kwangkeun Yi

Programming Research Laboratory, School of Computer Sci. & Eng., Seoul National University

{msjin○,kwang}@ropas.snu.ac.kr

요 약

이 논문에서는 다단계 프로그램의 실행흐름을 타입을 기반으로 분석하는 방법을 제시하고 있다. 다단계 프로그램이란 실행 중 코드를 만들고 실행할 수 있는 언어를 말한다. 본 논문에서는 기존의 고차 함수형 언어의 실행흐름을 분석하는 방법을 다단계 언어를 위해서 확장한 방법을 제시하고 있다. 이 분석은 한 수식이 평가될 때 호출될 수 있는 함수들의 집합을 정적으로 분석하여 다단계 프로그램의 실행흐름 정보를 제공한다.

## 1. 서 론

실행 흐름분석(control flow analysis)은 컴파일러 최적화에 필요한 기본적인 정보[1]이다. 그러나 ML같은 함수형 언어의 실행흐름을 결정하는 것은 쉬운 일이 아니다. 함수형 언어에서는 함수가 다른 함수의 인자로 전달되거나 혹은 함수호출의 결과가 또 다른 함수 일 수도 있기 때문이다.

다단계 프로그래밍[2]의 경우에는 실행 흐름의 분석이 더욱 복잡하다. 이 언어에서는 함수뿐만이 아니라 코드 또한 값으로 존재할 수 있기 때문이다. 이 코드 값은 함수일 수도 있으며 이러한 코드 값은 함수의 인자로 사용되거나 함수의 결과일 수도 있고, 새로운 코드를 만들거나 코드로써 수행될 수도 있다.

### 1.1 다단계 프로그램(multi-staged language)에서의 실행흐름.

실행흐름 분석은 주어진 프로그램의 실행흐름을 정적으로 분석하는 것이다. 이 논문에서는 실행흐름의 정의를 [3]에 정의된 개념을 따라 정의한다. 여기서 실행흐름이라고 하는 것은 식(expression)을 계산하는 동안 함수 적용(function application)이 일어나는 함수들의 집합으로 정의한다.

아래와 같은 다단계 프로그램이 있다고 하자. 이해를 위해서 Lisp[4]과 같은 구문을 사용한다. eval 은 코드를 실행하고, ` 는 코드를 만들어낸다. 또한 각 함수를 구분하기 위해서 함수마다 유일한 라벨(label)이 부여되어 있다.

```
let  g = `(λ1a. a 1)
     h = λ2b. b
in   (λ3c. (eval c) h ) g
```

밑줄이 쳐진 식에서 c는 실행 중 g 의 값인 코드 `(λ1a.a 1)로 치환된다. 따라서 **eval** c 를 실행하게 되면 그 결과로써 함수 (λ1a.a 1)가 된다. 그러면 함수식 (λ1a.a 1)를 h에 적용하게 되고 실행흐름은 함수 (λ1a.a 1)의 몸체(body) a 1로 전달 된다. 다시 a 1을 실행하게 될 때 a는 h의 값인 함수 (λ2b.b)로 치환된다. 따라서 함수 적용식 (a 1)은 실행흐름을 함수 (λ2b.b)의 몸체로 옮기게 된다. 따라서 함수의 실행흐름은 {1,2}로 기록된다. 이는 밑줄이 쳐진 식을 수행할 때 호출되는 함수는 라벨이 1인 함수와 라벨이 2인 함수란 뜻이다.

본 논문에서는 위와 같은 실행흐름-식을 실행했을 때 함수적용이 일어나는 함수들-을 분석하기 위해서 타입 시스템을 기반으로 한 효과 시스템(effect system)을 이용한다. [3]에서는 일반적인 함수형 언어의 실행흐름을 분석하기 위해서 타입을 기반으로 한 효과 시스템이 사용되었다. 본 논문에서는 [3]에서 사용된 타입 기반의 효과 시스템을 다단계 프로그램을 위해서 확장 하였다.

## 2. 분석 대상 언어

본 논문의 분석 대상이 되는 언어는 다단계 프로그래밍 언어[2]이다. 이 언어는 일반적인 람다 계산식(lambda calculus)에 실행 중 코드를 조작하기 위한 언어 생성자가 추가되어 있다. 이 언어의 핵심문법

구조(abstract syntacx)는 그림1 에 정의 되어 있다.

$$\begin{array}{rcll} \text{Exprs}\ e & \in & \text{Exp} & \rightarrow\ c \mid x \\ & & & \mid\ \lambda_l x.e \mid e_1 e_2 \\ & & & \mid\ \texttt{lift}\ e \mid \texttt{box}\ e \mid \texttt{unbox}_k e\ (k \geq 0) \end{array}$$

그림 1 핵심문법 구조

실행 중 코드를 조작하기 위한 추가적인 언어 생성자에는 lift, box, $\text{unbox}_k$ 가 있다. lift 는 값을 코드로 만들고 box는 코드를 만들어 낸다. 마지막으로 $\text{unbox}_k$ 는 만들어진 코드 값들을 이용해서 새로운 코드 값을 만들어 내거나(k > 0), 코드 값을 실행한다(k = 0). box, $\text{unbox}_0$, $\text{unbox}_k$는 각각 Lisp 매크로의 매크로 생성자(`), 매크로 실행(eval), 매크로 치환(,)에 해당한다. 추가적으로 함수들을 구분하기 위해서 코드상의 함수들은 유일한 라벨(label)을 가지게 된다.

### 3. 동적인 의미구조(Dynamic Semantics)

본 논문의 분석대상이 되는 언어의 동적인 의미구조(dynamic semantics)는 계산문맥(evaluation context)를 이용한 실행과정을 생각하는 의미구조(small-step operational semantics)로 정의된다. 본 논문에서 정의된 의미구조에서는 실행 중 발생하는 실행흐름(control-flow) 정보를 포함하도록 정의되어 있다.

이 언어에서는 계산의 단계가 2가지로 나뉜다. 0단계에서의 일반적인 함수호출, 코드를 실행하거나 값을 코드로 바꾸는 계산을 수행한다. 0 보다 단계가 높을 때에는 코드를 바꾸는 계산만을 수행한다. 단계는 box를 통해서 단계가 1만큼 올라가고 $\text{unbox}_k$를 통해서 k 단계만큼 내려간다.

값은 레덱스(redex)가 없는 식이다. 다단계 프로그램에서 값은 다단계 값(multi-staged value)라고 불리는데 이는 각 단계(stage)마다 값들이 존재하기 때문이다. 각 $n$ 단계의 값들을 $Val^n$ 에 속한다. 실행흐름 추적(control-flow trace, $Traces$)은 함수 라벨(label)들의 집합으로, 실행 중에 함수적용이 일어나는 함수들의 라벨(label)을 모은 집합이다. 이 실행흐름 추적은 이후 타입 시스템의 의해서 정적으로 분석되고 이때 함수들간의 적용 순서를 고려하지 않기 때문에 집합으로 정의하였다. 다단계 값들과 실행흐름 추적은 그림2 에 정의 되어 있다.

$$\begin{array}{rcll} v^n & \in & Val^n & \rightarrow\ c \mid \lambda_l x.v^1 \mid \texttt{box}\ v^1 & \text{if } n = 0 \\ & & & \rightarrow\ c \mid \lambda_l x.v^n \mid v_1^n v_2^n \mid \texttt{lift}\ v^n & \text{if } n > k \geq 0 \\ & & & \mid\ \texttt{box}\ v^{n+1} \mid \texttt{unbox}_k v^{n-k} & \\ & & FLabel & = \texttt{set of function labels} & \\ \varphi & \in & Traces & = 2^{FLabel} & \end{array}$$

그림 2 다단계 값 과 실행흐름 추적

계산 문맥(evaluation context)은 식의 레덱스를 정의한다. $C^n$ 은 $n$ 단계에서의 계산문맥이고 $[]^m$ 단계에서의 레덱스를 가지게 된다. $C^n$ 은 레덱스가 어느 단계에서 실행되는지를 추적한다. 계산 문맥(evaluation context)은 그림3 에 정의 되어 있다.

$$\begin{array}{rcll} \mathcal{C}^n & \in & \text{EC}^n & \rightarrow\ [\cdot]^n \mid \mathcal{C}^n e \mid v^n \mathcal{C}^n \mid \texttt{lift}\ \mathcal{C}^n & \\ & & & \mid\ \texttt{box}\ \mathcal{C}^{n+1} \mid \texttt{unbox}_k \mathcal{C}^{n-k} & \text{if } n \geq k \geq 0 \\ & & & \mid\ \lambda_l x.\mathcal{C}^n & \text{if } n > 0 \end{array}$$

그림 3 계산 문맥(evaluation context)

레덱스는 계산이 가능한 (reducible) 식이다. 0단계에서 레덱스는 일반적인 함수 호출이거나 0단계의 값을 코드로 바꾸거나(lift) 코드를 실행한다($\text{unbox}_0$). 0 보다 큰 단계에서는 기존의 코드를 이용하여 새로운 코드를 만드는 계산만이 발생한다. 레덱스는 그림4에 정의 되어 있다.

$$\begin{array}{rcll} r^n & \in & \text{Redexes} & \rightarrow\ v_1^0 v_2^0 \mid \texttt{lift}\ v^0 & \text{if } n = 0 \\ & & & \rightarrow\ \texttt{unbox}_n v^0 & \text{if } n \geq 0 \end{array}$$

그림 4 레덱스(redex)

레덱스가 변하는 관계(reduction relation)는 다음과 같다.

$$r^n \overset{n}{\hookrightarrow} e, \varphi$$

이는 레덱스 $r^n$ 이 $n$ 단계에서 식 $e$로 변하고, 이 때 실행흐름 추적($Traces$)으로 $\varphi$을 가진다는 뜻이다. 레덱스가 변하는 규칙은 그림5 에 정의 되어 있다.

(EAPP) 는 0 단계에서 함수 호출을 나타낸다. 이 때 함수의 형식인자(formal parameter) $x$를 함수의 실제인자(actual parameter) $v$ 로 치환하여 함수의 몸체를 실행하게 된다. 형식인자를 실제 인자로 치환하는 규칙은 그림6에 정의되어 있다. 이 때 함수 호출이 발생하게 되므로 호출되는 함수의 라벨을 실행흐름 추적으로 기록하게 된다.

(ELIFT) 는 0 단계에서 0단계의 값을 코드로 바꾸는 계산을 수행하게 된다.

(EUNBOX) 는 2가지 역할을 한다. 0단계에서는 1단계의

코드 값을 0단계에서 실행되게 한다. 이는 Lisp의 매크로 실행(eval)과 같은 역할을 한다. $n$ 단계($n$>0)에서는 $\text{unbox}_n$에 의해서 같은 단계가 내려가서 0단계에서 코드 값(box $v^1$)이 $n$단계의 값으로 바뀌게 된다. 이는 Lisp의 매크로 치환(,) 과 같은 역할을 한다.

계산관계(evaluation relation)는 다음과 같다.

$$e, \varphi \xrightarrow{n} e', \varphi'$$

이는 식 $e$ 가 실행흐름 $\varphi$을 가지고 있고 $e$'으로 계산될 때 실행 $\varphi$' 흐름 을 가진다는 뜻이다. 계산관계는 그림5에 정의되어 있다.

$$\begin{array}{ll}
\text{(EAPP)} & (\lambda_l x.e)v^0 \xrightarrow{0} [x^0 \overset{0}{\mapsto} v^0]e, \{l\} \\
\text{(ELIFT)} & \texttt{lift}\ v^0 \xrightarrow{0} \texttt{box}\ v^0, \emptyset \\
\text{(EUNBOX)} & \text{unbox}_n(\texttt{box}\ v^1) \xrightarrow{n} v^1, \emptyset \\
\text{(ECONTEXT)} & \dfrac{r^m \xrightarrow{m} e, \varphi'}{\mathcal{C}^n[r^m]^m, \varphi \xrightarrow{n} \mathcal{C}^n[e]^m, \varphi \cup \varphi'}
\end{array}$$

그림 5 레벡스의 변환관계(reduction relation)와 계산관계(evaluation relation)

$$\begin{array}{lcll}
[x \overset{n}{\mapsto} v]c & = & c & \\
[x \overset{n}{\mapsto} v]y & = & v & \textit{if } x = y \textit{ and } n = 0 \\
 & = & y & \textit{otherwise} \\
[x \overset{n}{\mapsto} v]\lambda_l y.e & = & \lambda_l y.e & \textit{if } x = y \textit{ and } n = 0 \\
 & = & \lambda_l y.[x \overset{n}{\mapsto} v]e & \textit{otherwise} \\
[x \overset{n}{\mapsto} v](e_1 e_2) & = & ([x \overset{n}{\mapsto} v]e_1)([x \overset{n}{\mapsto} v]e_2) & \\
[x \overset{n}{\mapsto} v](\texttt{box}\ e) & = & \texttt{box}\ ([x \overset{n+1}{\mapsto} v]e) & \\
[x \overset{n}{\mapsto} v](\texttt{unbox}_k e) & = & \texttt{unbox}_k([x \overset{n-k}{\mapsto} v]e) & \\
[x \overset{n}{\mapsto} v](\texttt{lift}\ e) & = & \texttt{lift}\ ([x \overset{n}{\mapsto} v]e) &
\end{array}$$

그림 6 치환 규칙 ($x$를 $n$ 단계에서 $v$로 치환)

## 4. 정적인 의미구조(Static Semantics)

분석의 대상이 되는 언어의 정적인 의미구조는 식의 타입과 그 식이 실행될 때 호출되는 함수들의 라벨을 결정한다. 컨트롤 $\delta$ 은 동적인 의미에서 정의된 실행흐름 추적 $\varphi$ 을 요약하여 분석한 것이다. 즉 $\delta$ 는 실제 실행흐름을 추적하는 $\varphi$ 를 포함한다.

$$\begin{array}{llll}
A, B & \in & \textit{Type} & A, B \rightarrow \iota \\
 & & & \mid\ A \xrightarrow{\delta} B \\
 & & & \mid\ \square(\Gamma \triangleright A, \delta) \\
\Gamma & \in & \textit{TyEnv} & = \textit{Var} \xrightarrow{fin} \textit{Type} \\
\delta & \in & \textit{Control} & \delta \rightarrow \emptyset \mid \{l\} \mid \delta \cup \delta'
\end{array}$$

그림 7 타입

프로그램의 식(expression)이 가질 수 있는 타입으로는 상수, 함수, 코드 타입이 있다. $\iota$ 는 상수 타입이고, $A \xrightarrow{\delta} B$ 은 함수 타입으로 잠재적인 효과(latent effect)로 $\delta$을 가지고 있다. 잠재적인 효과(latent effect) $\delta$의 의미는 이 함수 타입의 함수가 호출되어 실행되면 호출될 수 있는 모든 함수들의 라벨(label)을 뜻한다. $\square(\Gamma \triangleright A, \delta)$ 은 코드 타입니다. 코드 타입 안에 있는 타입 환경(type environment)은 코드의 자유변수가 어떤 타입을 가져야 하는 지를 말한다. 코드 타입 안의 타입 $A$ 는 이 타입의 코드가 0 단계에서 실행되면 그 결과로써 $A$라는 타입의 값을 가진다는 뜻이다. 마지막으로 코드 타입안에 있는 잠재적인 효과(latent effect) $\delta$ 는 이 코드가 0 단계에서 실행될 때 호출될 수 있는 함수들은 $\delta$ 에 속한다는 뜻이다.

### 4.1 타입 규칙

식의 타입은 다음의 타입 규칙을 통해서 결정된다.

$$\Gamma_0 \cdots \Gamma_n \vdash e : A, \delta_n \cdots \delta_0$$

이는 식 $e$가 $n$ 단계에서 타입환경 $\Gamma_0 \cdots \Gamma_n$ 을 가지고 실행되면, $A$ 타입의 값을 가지게 되고, 효과(effect)로는 $\delta_n \cdots \delta_0$ 을 가진다는 뜻이다. $\delta_n \cdots \delta_0$은 두 가지 종류로 나누어 진다.

0 단계 효과($\delta_0$)의 의미는 식 $e$가 $n$ 단계에서 실행될 때 0 단계에서 발생하는 효과를 뜻한다. 따라서 $\delta_0$는 $n$ 단계에서 실행될 때 호출될 수 있는 함수들을 뜻한다. $i$ 단계($i$ > 0) 효과 $\delta_i$ 의 의미는 다음과 같다. 식 $e$ 의 하위 식(sub expression)이 실행될 때 단계가 낮아 질 수 있는데 이 때 낮아진 각 단계에서의 효과를 표시한다. 따라서 이 효과들은 $e$ 가 $n$ 단계에서 실행될 때 호출되는 함수들을 뜻하는 것이 아니라 $e$를 감싸고 있는 $\text{unbox}_0$들에 의해서 단계가 ($n$-$i$)만큼 낮아져서 실행될 때 발생하는 효과(effect)를 기록하고 있는 것이다.

타입 규칙은 그림8 에 정의되어 있고, 각 타입 규칙에 대한 설명은 아래와 같다.

*TSCONST* 는 상수(constant) 에 대한 타입 규칙이고 효과를 가지고 있지 않다. *TSVAR* 는 같은 단계의 타입환경에 변수의 타입에 대한 가정이 있어야 한다는 뜻이다.

$$TSCONST \quad \frac{}{\Gamma_0 \cdots \Gamma_n \vdash c : \iota, \emptyset_n \cdots \emptyset_0}$$

$$TSVAR \quad \frac{\Gamma_n(x) = A}{\Gamma_0 \cdots \Gamma_n \vdash x : A, \emptyset_n \cdots \emptyset_0}$$

$$TSABS \quad \frac{\Gamma_0 \cdots \Gamma_n + x : A \vdash e : B, \delta_n \cdots \delta_0}{\Gamma_0 \cdots \Gamma_n \vdash \lambda_l x.e : A \xrightarrow{\{l\} \cup \delta_n} B, \emptyset\delta_{n-1} \cdots \delta_0}$$

$$TSAPP \quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e_1 : A \xrightarrow{\delta_1} B, \delta_n^1 \cdots \delta_0^1 \quad \Gamma_0 \cdots \Gamma_n \vdash e_2 : A, \delta_n^2 \cdots \delta_0^2}{\Gamma_0 \cdots \Gamma_n \vdash e_1 e_2 : B, (\delta_1 \cup \delta_n^1 \cup \delta_n^2) \cdots \delta_0^1 \cup \delta_0^2}$$

$$TSBOX \quad \frac{\Gamma_0 \cdots \Gamma_n \Gamma \vdash e : A, \delta\delta_n \cdots \delta_0}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{box}\ e : \square(\Gamma \triangleright A, \delta), \delta_n \cdots \delta_0}$$

$$TSUNBOX \quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : \square(\Gamma_{n+k} \triangleright A, \delta), \delta_n \cdots \delta_0 \qquad k > 0}{\Gamma_0 \cdots \Gamma_n \cdots \Gamma_{n+k} \vdash \texttt{unbox}_k e : A, \delta\emptyset_{n+k-1} \cdots \emptyset_{n+1}\delta_n \cdots \delta_0}$$

$$TSEVAL \quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : \square(\emptyset \triangleright A, \delta), \delta_n \cdots \delta_0}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{unbox}_0 e : A, \delta \cup \delta_n \cdots \delta_0}$$

$$TSLIFT \quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : A, \delta_n \cdots \delta_0}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{lift}\ e : \square(\Gamma \triangleright A, \emptyset), \delta_n \cdots \delta_0}$$

$$TSSUB \quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : A, \delta_n \cdots \delta_0 \quad \delta_i \subseteq \delta_i', i \in [0..n]}{\Gamma_0 \cdots \Gamma_n \vdash e : A, \delta_n' \cdots \delta_0'}$$

그림 8 타입 규칙

*TSABS* 는 함수 인자의 타입이 $A$ 라고 가정하면 결과의 타입은 함수의 몸체의 타입 $B$가 된다는 것이다. 그리고 함수의 몸체의 효과가 함수의 효과가 된다. 그러나 함수의 몸체의 효과 중 현재 단계의 효과 $\delta_n$ 는 이 함수가 실제 호출 될 때 발생하므로 함수 타입에 잠재적인 효과로 기록된다.

$$TSABS \quad \frac{\Gamma_0 \cdots \Gamma_n + x : A \vdash e : B, \delta_n \cdots \delta_0}{\Gamma_0 \cdots \Gamma_n \vdash \lambda_l x.e : A \xrightarrow{\{l\} \cup \delta_n} B, \emptyset\delta_{n-1} \cdots \delta_0}$$

*TSAPP* 는 $e_1$ 의 타입이 함수 타입이고 $e_2$ 가 함수의 인자의 타입이면 함수 호출식의 타입은 함수의 결과 타입이 된다는 의미이고, 효과는 각 단계의 효과들을 합친 것이다. 이 때 현재 단계의 효과에는 함수 타입에 기록된 잠재적인 효과들도 합쳐진다. 함수 타입에 기록된 잠재적인 효과들은 함수가 호출될 때 발생하는 효과를 기록하고 있기 때문이다.

$$TSAPP \quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e_1 : A \xrightarrow{\delta_1} B, \delta_n^1 \cdots \delta_0^1 \quad \Gamma_0 \cdots \Gamma_n \vdash e_2 : A, \delta_n^2 \cdots \delta_0^2}{\Gamma_0 \cdots \Gamma_n \vdash e_1 e_2 : B, (\delta_1 \cup \delta_n^1 \cup \delta_n^2) \cdots \delta_0^1 \cup \delta_0^2}$$

*TSBOX* 는 식 $e$ 가 하나 높은 단계에서 실행되고 그 때 타입이 $A$이면, 전체 식은 코드 타입이 된다는 의미이다. 이 때 한 단계 위에서 발생하는 효과는 코드 타입에 잠재적인 효과로 기록되고, 이 효과는 $\texttt{unbox}_0$ 실행되거나 $\texttt{unbox}_k$(k>0) 에 의해서 다른 코드 속으로 들어갈 때 사용된다.

$$TSBOX \quad \frac{\Gamma_0 \cdots \Gamma_n \Gamma \vdash e : A, \delta\delta_n \cdots \delta_0}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{box}\ e : \square(\Gamma \triangleright A, \delta), \delta_n \cdots \delta_0}$$

*TSUNBOX* 는 현재 단계보다 낮은 단계에서 식 $e$ 가 실행되고 코드 타입 $\square(\Gamma_{n+k} \triangleright A, \delta)$일 경우 전체 식은 $A$ 타입이 된다는 것이다. 이는 $e$ 가 실행되어서 코드값이 되면 전체 식은 이 코드로 바뀌기 때문이다. 이 때 $e$의 코드 타입에 있는 잠재적인 효과는 현재 단계의 효과로 바뀌게 된다.

$$TSUNBOX \quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : \square(\Gamma_{n+k} \triangleright A, \delta), \delta_n \cdots \delta_0 \qquad k > 0}{\Gamma_0 \cdots \Gamma_n \cdots \Gamma_{n+k} \vdash \texttt{unbox}_k e : A, \delta\emptyset_{n+k-1} \cdots \emptyset_{n+1}\delta_n \cdots \delta_0}$$

*TSEVAL* 은 $e$ 가 코드 타입이고 코드 속에 자유변수가 없는 닫힌 코드이면 전체 식은 $A$ 타입이 된다는 것이다. 이는 $\texttt{unbox}_0$ 에 의해서 $e$가 실행된 결과인 코드를 다시 실행하기 때문이다. 따라서 이 때의 효과는 $e$의 코드 타입에 있는 잠재적인 효과를 현재 단계의 효과에 포함하게 된다.

$$TSEVAL \quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : \square(\emptyset \triangleright A, \delta), \delta_n \cdots \delta_0}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{unbox}_0 e : A, \delta \cup \delta_n \cdots \delta_0}$$

*TSLIFT* 는 $e$가 $A$타입이면 $e$의 결과를 코드로 만들면 같은 타입의 코드 타입이 된다는 것이다. 또한 이 때 코드 타입 안의 환경은 임의의 환경을 가져도 된다는 뜻이다.

$$TSLIFT \quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : A, \delta_n \cdots \delta_0}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{lift}\ e : \square(\Gamma \triangleright A, \emptyset), \delta_n \cdots \delta_0}$$

*TSSUB* 는 임의의 식이 더 많은 효과를 가질 수 있도록 해준다. 이 타입 룰은 효과 때문에 타입이 맞지 않아 전체 식의 타입을 결정할 수 없을 때 타입을 맞추기 위해서 사용할 수 있다.

$$TSSUB \quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : A, \delta_n \cdots \delta_0 \quad \delta_i \subseteq \delta_i', i \in [0..n]}{\Gamma_0 \cdots \Gamma_n \vdash e : A, \delta_n' \cdots \delta_0'}$$

### 4.2 안전성

타입 시스템의 안전성은 타입을 가지는 식은 실행 중 타입 에러를 발생시키지 않는다는 것을 보장하는 것이다. 이는 보조정리1(*Progress*)과 보조정리2(*Subject Reduction*)를 통해서 증명된다.

**보조정리 1.** (*Progress*)

*If* $\emptyset\Gamma_1 \cdots \Gamma_n \vdash e : A, \delta_n \cdots \delta_0$,
*then* $e$ *is in* $Val^n$ *or else* $e, \varphi \xrightarrow{n} e', \varphi'$.

보조정리1(*Progress*)은 식이 타입을 가지면 그 식은 값에 속하거나 진행을 한다는 뜻이다.

**보조정리 2.** (*Subject Reduction*)

*If* $\emptyset\Gamma_1 \cdots \Gamma_n \vdash e : A, \delta_n \cdots \delta_0$
*and* $e, \varphi \xrightarrow{n} e', \varphi'$ *where* $\varphi \subseteq \delta_0$
*then* $\emptyset\Gamma_1 \cdots \Gamma_n \vdash e' : A, \delta_n \cdots \delta_0, \varphi' \subseteq \delta_0$

보조정리2 (*Subject Reduction*)은 타입을 가지고 진행을 하면, 진행을 하고 난 식 또한 같은 타입을

가진다는 뜻이다. 또한 진행하고 난 이후의 식이 가지는 효과는 원래 식의 효과와 같고 식이 진행 할 때 의 발생하는 실행흐름 추적은 원래 식의 효과에 포함된다는 뜻이다.

타입 시스템을 기반으로 분석한 효과의 안전성은 아래의 정리로 증명된다.

정리 1.

$$\text{If } \emptyset\Gamma_1 \ldots \Gamma_n \vdash e : A, \delta_n \cdots \delta_0 \text{ and } e, \emptyset \xrightarrow{n}{}^{*} v, \varphi,$$
$$\text{then } \emptyset\Gamma_1 \ldots \Gamma_n \vdash v : A, \delta_n \cdots \delta_0 \text{ and } \varphi \subseteq \delta_0.$$

정리 1은 식이 타입을 가지고 계속 진행하여 값이 되면, 그 값은 원래 식과 같은 타입을 가지고 실행 중 발생했던 실행흐름 추적은 모두 원래 식의 효과에 포함된다는 것이다. 즉, 실행하면서 호출되었던 함수는 모두 타입시스템으로 분석된 효과에 포함된다는 뜻이다.

위의 보조 정리1,2와 정리1에 대한 증명은 http://ropas.snu.ac.kr/~msjin/cfa_msp.pdf 에서 찾아 볼 수 있다.

## 5. 결론 및 향후 과제

본 논문에서는 다단계 프로그램에 대한 실행흐름 분석을 타입을 기반으로 한 효과 시스템으로 분석하는 방법을 보이고 그 방법의 정확성에 대한 증명을 수행하였다. 또한 본 논문의 응용으로 실행 흐름 분석 정보를 이용하여 [3]에서 제시한 이스케이프 분석(escape analysis)을 이용한 함수값(closure) 할당 최적화를 다단계 프로그램에 적용할 수 있다.

향후 과제로는 기반이 되는 타입 시스템을 더욱 정교한 타입 시스템으로 확장하는 것이다. 서브타이핑 시스템(subtyping system)과 let-다형성 (let-polymorphism)을 지원하도록 확장하면 본 논문에서 사용하는 단순 타입 시스템(simple type system)보다 더욱 정확한 분석을 수행할 수 있고, 더 많은 프로그램을 분석 할 수 있을 것이다.

## 6. 참고 문헌

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers: principles, technniques, and tools. *Addison-Wesley Longman Publishing Co., Inc*, 1986.

[2] 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*, 257-268, 2006.

[3] Yan-Mei Tang. Effect System and Abstract Interpretation for Control Flow Analysis. PhD thesis, *ECOLE NATIONALE SUPERIEURE DES MINES DE PARIS*, 1994.

[4] Paul Graham. *On Lisp: an advanced technique for Common Lisp*. Prentice Hall, 1994.