Eval-Apply 모델의 STGM에 기반하여 지연 계산 함수형 프로그램을 자바로 컴파일하는 기법

(Compiling Lazy Functional Programs to Java on the basis of Spineless Tagless G-Machine with Eval-Apply Model)

남 병 규 † 최 광 훈 †† 한 태 숙 †††

(Byeong-Gyu Nam) (Kwanghoon Choi) (Taisook Han)

요 약 최근에 지연 계산 함수형 언어를 자바 프로그램으로 변환함으로써 지연 계산 함수형 언어 프로그램에 대해 코드 이동성을 제공하려는 연구가 있었다. 이러한 연구들은 자바와 지연 계산형 함수형 언어의 추상 기계가 가지는 구조적 유사성에 바탕을 두고 있다. 지연 계산 함수형 언어에 대한 추상 기계인 STGM(Spineless Tagless G-Machine)과 자바 언어에 대한 추상 기계인 JVM(Java Virtual Machine)은 기억장소 재활용 체계와 스택 기계 구조를 가진다는 점에서 공통된 특징을 가지고 있다. 그러나 현재까지의 지연 계산 함수형 언어로부터 자바로의 변환 구조는 이와 같은 추상 기계 구조상의 공통점을 충분히 이용하지 못하였다.

본 논문에서는 STGM의 계산 모델을 eval-apply 모델로 새로이 정의함으로써 STGM과 JVM의 공통점을 충분히 이용하는 새로운 변환 구도를 제안한다. 새로이 제안된 변환 구도에서는 자바 스택(Java Virtual Machine Stack)을 사용하여 함수 계산을 수행하도록 함으로써 스택 시뮬레이션으로 인해 나타나는 자바에서의 배열 접근 부담을 제거하였다.

본 논문의 변환 구도에 의해 자바로 변환된 벤치마크 프로그램들은 기존의 변환 구도에 의해 변환된 경우보다 JDK 1.3에서 빠르게 동작한다.

키워드 : 프로그래밍 언어, 함수형 언어, 자바, 컴파일러, 지연 계산, 추상 기계

Abstract Recently there have been a number of researches to provide code mobility to lazy functional language (LFL) programs by translating LFL programs to Java programs. These approaches are basically based on architectural similarities between abstract machines of LFLs and Java. The abstract machines of LFLs and Java programming language, Spineless Tagless G-Machine(STGM) and Java Virtual Machine(JVM) respectively, share important common features such as built-in garbage collector and stack machine architecture. Thus, we can provide code mobility to LFLs by translating LFLs to Java utilizing these common features.

In this paper, we propose a new translation scheme which fully utilizes architectural common features between STGM and JVM. By redefining STGM as an eval-apply evaluation model, we have defined a new translation scheme which utilizes Java Virtual Machine Stack for function evaluation and totally eliminates stack simulation which causes array manipulation overhead in Java.

Benchmark programs translated to Java programs by our translation scheme run faster on JDK 1.3 than those translated by the previous schemes.

Key words : programming language, functional language, Java, Compiler, lazy evaluation, abstract machine, eval-apply model, STGM, JVM

· 본 연구는 첨단정보기술 연구센터를 통하여 과학재단의 지원을 받음.

† 비 회 원 : 한국전자통신연구원 시스템연결망연구팀
bgnam@etri.re.kr

†† 비 회 원 : 한국과학기술원 전산학과
khchoi@kaist.ac.kr

††† 종신회원 : 한국과학기술원 전산학과 교수
han@mail.kaist.ac.kr

논문접수 : 2001년 6월 27일
심사완료 : 2002년 1월 29일

1. 서 론

지연 계산 함수형 언어에서의 컴파일 기법들은 추상 기계(abstract machine)를 중간 단계에 도입해서 사용해 왔다[1]. 이러한 추상 기계의 개념은 현재 자바 언어에서 도입하고 있는 가상 기계인 Java Virtual Machine (JVM)[2][3]과 유사한 개념이지만 지연 계산 함수형 프로그램에게는 현재의 자바와 같은 코드의 이동성이 제공되지 못하였다.

그러나 JVM의 특징인 스택 머신 아키텍처와 기억장소 재활용 체계(garbage collector)는 현재 함수형 언어들에 대한 추상 기계들과 특징을 같이 하므로 이러한 특징을 이용하면 JVM을 통하여 지연 계산 함수형 언어 프로그램에 코드 이동성을 제공하는 것이 가능해진다. 또한 이 경우 많은 라이브러리를 제공하는 자바 언어와의 혼합 프로그래밍도 가능해지게 된다[4].

본 논문에서는 지연 계산 함수형 프로그램에 코드 이동성을 제공하기 위해 지연 계산 함수형 언어에 대한 대표적 추상 기계인 Spineless Tagless G-Machine (STGM)[1]으로부터 JVM[2][3]으로의 매핑 구도를 제안한 후 효율성을 실험한다. 이를 위해 함수형 언어의 추상 기계에 대한 계산 모델인 eval-apply 모델[1][5]에 기반하여 STGM-to-JVM 컴파일러를 제안한 후 벤치 마킹을 통해 기존의 push-enter 모델[1][5]에 기반한 컴파일러와 성능을 비교 분석한다.

본 논문의 전체적인 구조는 다음과 같다. 2장에서는 함수형 언어를 JVM에서 수행하도록 하는 기존의 연구들에 관해서 살펴보고 본 논문과 관련된 두 가지의 추상 기계인 STGM과 JVM을 각각 소개한다. 3장에서는 eval-apply 모델에 기반한 STG-to-Java 컴파일러를 제안한다. 4장에서는 실험을 통해 eval-apply 모델에 기반한 컴파일러의 성능을 기존의 push-enter 모델에 기반한 컴파일러와 비교 분석하고 5장에서 결론을 맺는다.

2. 관련 연구

이 절에서는 본 논문에서 다루게 될 지연 계산 함수형 언어에 대한 추상 기계인 Spineless Tagless G-Machine(STGM)과 자바 언어의 가상 기계인 Java Virtual Machine(JVM)에 대해서 각각 알아본다. 그리고 함수형 언어를 JVM 상에서 수행하기 위한 기존의 연구들에 관해서 살펴본다.

2.1 Spineless Tagless G-Machine

STGM은 지연 계산 함수형 언어를 수행하는 추상 기계로서 그래프 축약(graph reduction)을 위해 스택 머신 아키텍처를 가지며 리덕션이 끝난 후 갱신이 되는 스파인 없음(spineless)의 특징과 값(value)과 썽크(thunk)가 하나의 형태로 포함되는 태그 없음(tagless)의 특징을 갖는다.

2.2 Application에 대한 컴파일 방법

2.2.1 Push-Enter Model

이 모델에서는 현재 이용 가능한 함수의 인자들을 모두 전역 스택에 넣은 후 계산해야 할 함수의 코드로 이동 한다. 그리고 한 번 다른 클로저(closure)로 이동한 후에는 다시 이전의 환경으로 돌아오지 않는 종단 호출(tail-call)을 이용한다. 그러므로 이 모델은 인자를 스택에 넣을 때의 환경과 함수가 인자에 적용될 때의 환경이 다르다는 특징을 가진다. 따라서 다른 클로저로 이동하기 전에는 연속(continuation) 즉, 앞으로 수행되어야 할 내용과 전달되어야 할 인자(argument)가 다른 클로저에서 접근 가능한 전역 스택에 저장되어야 한다.

2.2.2 Eval-Apply Model

이 모델에서는 함수를 먼저 계산한 후 인자들이 있는 환경으로 돌아와서 함수를 인자에 적용(apply)한다. 그러므로 함수가 계산되기 전의 환경과 함수가 인자에 적용될 때의 환경이 같다는 특징을 가진다. 특히 이 모델에서는 함수 적용(function application)이 하나의 독립된 연산의 의미를 가진다. 즉, 함수 적용식 (f x y)는 f를 계산하여 얻은 클로저를 x에 적용하고 또 그 결과로 얻은 클로저를 y에 적용하여 결과 클로저를 얻게 되므로 이 모델에서는 함수 적용이 하나의 연산이 되어 연산에 대한 결과 값의 보존을 위해 매번 함수 적용이 수행될 때마다 새로운 클로저가 생성되게 된다.

2.3 Java Virtual Machine

Java Virtual Machine(JVM)의 스택인 Java Virtual Machine Stack (JVM Stack)은 프레임(frame)들이 저장되는 장소이다. 프레임이란 메쏘드들의 수행 시 각 메쏘드들의 상태(state)를 저장하는 자료 구조로서 기존의 C와 같은 언어에서의 활성 레코드(activation record)에 해당한다[2].

메쏘드들 간의 호출이 발생할 경우 호출한 메쏘드의 상태를 담고 있는 프레임이 JVM Stack에 할당되게 된다. 이 때 각 프레임은 해당 메쏘드에 대한 실인자(actual parameter), 지역 변수(local variable) 그리고 메쏘드 수행에 대한 중간 계산 결과를 담고 있는 로컬 스택(local stack)을 포함한다[2].

힙(heap)은 자바 프로그램이 실행 중에 동적으로 메모리를 할당할 때 사용되는 공간으로서 클래스에 대한 실체(instance)인 객체(object)들이 이곳에 할당되며 이

들은 사용된 후 기억장소 재활용체계(garbage collection)에 의해 회수된다[2].

2.4 기존 연구

2.4.1 Push Enter 모델에 기반한 컴파일러

Push-enter 모델에 기반한 JVM으로의 매핑에 관한 연구는 Tullsen[6]과 Vernet[7] 그리고 CLH[4]에 의해 있었다. Push-enter 모델에 기반한 컴파일러에서는 전역 스택을 구현하기 위한 배열이 큰 부담이 되었다. 왜냐하면 자바에서의 배열에 대한 접근은 배열에 대한 경계 검사와 기억장소 유출(space leak) 방지를 위한 널(null) 할당 등의 부담을 갖는다.

Tullsen은 push-enter 모델의 STGM이 JVM으로 매핑될 때의 문제점들을 관찰하고 고차 함수의 구현에 대해서는 그 논문에서 정의한 apply 메쏘드를 사용함으로써 eval-apply 모델의 STGM에 기반하여 컴파일러를 설계하였다. 그러나 이들은 예제에 기반하여 설명하고 있으므로 컴파일 규칙을 제시하지 않았으며 실험 결과 또한 제시하지 않고 있다.

CLH[4]는 push-enter 모델에 기반한 STGM을 실행 과정이 클로저라는 보다 거시적 관점에서 정의된 L-machine으로 나타내고 이에 기반하여 STG-to-Java 컴파일러를 정의하고 있다. 그러므로 여기서에는 컴파일러가 2단계로 정의되어 있으며 STG-to-L-code 컴파일러와 L-code-to-Java 컴파일러로 나뉘어 정의되어 있다.

2.4.2 Eval Apply 모델에 기반한 컴파일러

Eval-apply 모델에 기반한 컴파일러에 대한 연구로는 $<\nu, G>$-machine에 기반한 연구가 있었다[8][9]. 여기에서는 $<\nu, G>$-machine의 각 프레임 내에 존재하는 로컬 스택들이 모여서 전체 스택을 구성하므로 push-enter 모델 기반 컴파일러에서의 배열에 대한 비용과 기억장소 유출(space leak) 등을 없앨 수 있었다. 이 컴파일러는 Hugs와 동등한 성능을 보였다.

이와 같은 특징을 가지는 eval-apply 모델에 기반하여 STGM을 정의한다면 스파인 없음과 태그 없음의 이점을 살림과 동시에 push-enter 모델에서 나타났던 문제점에 대한 해결책이 될 수 있을 것이다.

3. Eval-Apply 모델의 STGM

이 장에서는 eval-apply 모델에 기반하여 STGM을 제안한다. 이를 위해 클로저를 실행 과정으로 가지는 L-machine[4]을 eval-apply 모델에 기반하여 제안한다.

3.1 STG언어

이 장에서 원시언어(source language)로 사용되는 STG 언어는 STGM에 대한 기계어로서 제한된 형태의 함수형 언어이다. 그림 3.1은 STG의 문법을 나타낸 것이다.

x, y, z, w			변수
v	::=	$\lambda\overline{x}.e \mid c\ \overline{x}$	값
e	::=	$x_0\ \overline{x} \mid$ let $\overline{bind}$ in $e \mid$ case e of $\lambda z.alts$	식
b	::=	$v \mid e$	바운드 식
π	::=	$u \mid n$	갱신 플래그
$bind$	::=	$x \stackrel{\pi}{=} b$	바인딩
t	::=	$c \mid \lambda \mid$ default	태그
$alts$	::=	$\overline{alt}$	선택자들
alt	::=	$c\ \overline{x} \rightarrow e \mid$ default $\rightarrow e$	선택자
$decl$	::=	T $\overline{c_i\ n_i}$	타입 선언
prg	::=	$\overline{decl}$, let $\overline{bind}$ in $main$	프로그램

그림 3.1 STG 문법

표기법을 설명하면 다음과 같다. $\overline{obj_n}$은 $obj_1..obj_n(n \geq 0)$을 나타내며 ε은 공백 열을 나타낸다. 모든 바운드 식(bound expression)들은 불완전 정규형(weak head normal form : WHNF)에 대한 값 v를 나타내거나 식(expression)에 대한 썽크(thunk) e를 나타낸다. 또 모든 바인딩(binding)들은 썽크의 연산 결과에 대한 공유 여부를 결정하기 위해 갱신 표식(update flag) π를 가진다. **case** 식에서는 피검사식 e의 계산된 값이 태그로서 z에 바인드된다. 태그 λ는 생성자 태그 c와는 구별되는 람다 식(lambda abstraction)에 대한 태그이고 *default* 태그는 편의를 위해 도입되었다.

전체 STG 프로그램은 생성자에 대한 태그와 인자 수를 정의하는 타입 선언들, 그리고 **let** 식들로 이루어진다.

3.2 L-code

L-code의 각 명령어는 STGM의 각 동작을 나타낸다. 그림 3.2는 eval-apply 모델의 STGM에 대한 L-code의 문법을 나타낸 것이다.

x,y,z,w,t,s					레지스터
l					레이블
C	::=	**let** $\overline{l = C}$ **in** C			L-code
	|	CASE t $\overline{t \rightarrow \langle l, x_0\ \overline{x}\rangle}$			
	|	GETN (λx.C)	| GETNT (λx.λt.C)	| GETA ($\lambda\langle l, x_0\ \overline{x}\rangle$.C)	
	|	EVAL x	| APPLY f x	| RET x t	
	|	PUTC x $\stackrel{\pi}{=} \langle l, \overline{x}\rangle$ C	| GETC x ($\lambda\langle l, \overline{w}\rangle$.C)	| UPDC x $\langle l, \overline{w}\rangle$	
	|	ARGCK x n C_{mp} C	| STOP x		

그림 3.2 L-code의 문법

3.3 L-machine

L-machine은 3.2절에서 정의한 L-code를 수행하는 기계로서 L-code에 대한 축약 규칙(reduction rule)으로 정의되며 그 축약 규칙은 그림 3.4와 같다. 그림 3.3은 eval-apply 모델의 STGM에 대한 L-machine의 구성 요소를 나타내고 있다.

본 논문에서 정의하는 L-machine은 기계의 상태에 대한 패턴 매치에 의해 구동되며 기계의 상태는 C μs h로 나타난다. 레지스터 파일 μ는 각 레지스터로부터 주소, 레이블, 태그로의 매핑이고 스택 s는 코드 C와 레지스터 파일 μ로 이루어진 활성 레코드 [C, μ]의 열이다. 그리고 힙 h는 주소로부터 클로저로의 매핑 또는 레이블로부터 L-code로의 매핑이고 μ_0, s_0, h_0는 각각 비어있는 레지스터 파일, 스택, 힙을 나타낸다. 그리고 축약이 끝나면 결과 값의 주소와 L-machine의 마지막 힙에 대한 쌍 (x, h)를 결과로 낸다. Eval-apply 모델에서는 호출 상황(call context)을 저장하기 위해 스택이 사용된다. 그리고 이 스택에는 각 클로저의 호출 상황인 활성 레코드(activation record) [C, μ]들이 쌓이므로 인자와 연속을 저장하는 push-enter 모델의 스택과는 구별된다.

3.4 STG-to-L-code 컴파일러

이 절에서는 STG프로그램으로부터 eval-apply 모델의 L-code프로그램을 만들어내는 컴파일러를 정의한다. 이를 위해 여기서는 간접 지칭(indirection)에 대한 (*ind* w), 생성자(constructor)에 대한 ($c\,w$)를 STG에 추가한다. STG의 몇 가지 문법 요소에 대해 자유 변수 집합

$\langle l, \rho \rangle$	**클로저**
$\rho ::= \overline{x}$	**환경**
$\mu ::= \mu_0 \mid \mu[\mathrm{x} \mapsto x] \mid \mu[\mathrm{x} \mapsto l] \mid \mu[\mathrm{t} \mapsto t]$	**레지스터 파일**
$s ::= s_0 \mid [\mathrm{C}, \mu]\ s$	**스택**
$h ::= h_0 \mid h[x \mapsto \langle l, \rho \rangle] \mid h[l \mapsto \mathrm{C}]$	**힙**

그림 3.3 L-machine의 구성 요소

(let $\overline{l = \mathrm{C}}$ in C_0) $\mu\ s\ h$	$\Rightarrow \mathrm{C}_0\ \mu\ s\ h[\overline{l \mapsto \mathrm{C}}]$	
(GETN (λx.C)) $x\ \mu\ s\ h$	$\Rightarrow \mathrm{C}\ \mu[\mathrm{x} \mapsto x]\ s\ h$	
(GETNT (λx.λt.C) $x\ t\ \mu\ s\ h$	$\Rightarrow \mathrm{C}\ \mu[\mathrm{x} \mapsto x, \mathrm{t} \mapsto t]\ s\ h$	
(GETA ($\lambda\langle l, \overline{\mathrm{x}} \rangle$.C) $\langle l, \overline{x} \rangle\ \mu\ s\ h$	$\Rightarrow \mathrm{C}\ \mu[\overline{\mathrm{x} \mapsto x}]\ s\ h$	
(STOP x) $\mu\ s\ h$	$\Rightarrow (\mu(\mathrm{x}), h)$	
(CASE t $\overline{t_i \rightarrow \langle l_i, \overline{\mathrm{x}}_i \rangle}$) $\mu\ s\ h$	$\Rightarrow h(l_j)\ \langle l_j, \overline{\mu(\mathrm{x})}_i \rangle\ s\ h$	if $\exists j . \mu(\mathrm{t}) = t_j$
	$\Rightarrow h(l_d)\ \langle l_d, \overline{\mu(\mathrm{x})}_d \rangle\ s\ h$	otherwise[1]
(ARGCK x n C_{mp} C) $\mu\ s\ h$	$\Rightarrow \mathrm{C}\ \mu\ s\ h$	$\psi(\mathrm{x}, n)$[2]
	$\Rightarrow \mathrm{C}_{mp}\ \mu\ s\ h$	otherwise
(GETC x ($\lambda\langle l, \overline{\mathrm{w}} \rangle$.C)) $\mu\ s\ h$	$\Rightarrow \mathrm{C}\ \mu[\overline{\mathrm{w} \mapsto w}]\ s\ h$	$\theta(\mathrm{x}, l, \overline{w})$[3]
(PUTC x $\overline{=}$ $\langle l, \overline{\mathrm{w}} \rangle$ C) $\mu\ s\ h$	$\Rightarrow \mathrm{C}\ \mu'\ s\ h[x^* \mapsto \langle l, \overline{\mu'(\mathrm{w})} \rangle]$	$\mu' = \mu[\mathrm{x} \mapsto x^*]$
(UPDC x $\langle l, \overline{\mathrm{w}} \rangle$ C) $\mu\ s\ h$	$\Rightarrow \mathrm{C}\ \mu\ s\ h[\mu(\mathrm{x}) \mapsto \langle l, \overline{\mu(\mathrm{w})} \rangle]$	
(EVAL x C) $\mu\ s\ h$	$\Rightarrow h(l)\ \mu(\mathrm{x})\ \mu_0\ ([\mathrm{C}, \mu]\ s)\ h$	$\theta(\mathrm{x}, l)$[4]
(APPLY f x C) $\mu\ s\ h$	$\Rightarrow h(l)\ y^*\ \mu_0\ ([\mathrm{C}, \mu]\ s)\ h[y^* \mapsto \langle l, \rho\mu(\mathrm{x}) \rangle]$	$\theta(\mathrm{f}, l, \rho)$
(RET x t) $\mu\ ([\mathrm{C}', \mu']\ s)\ h$	$\Rightarrow \mathrm{C}'\ \mu(\mathrm{x})\ \mu(\mathrm{t})\ \mu'\ s\ h$	

1. default $\rightarrow \langle l_d, \mathrm{x}_0\ \overline{\mathrm{x}}_d \rangle$ 가 선택됨
2. $\psi(\mathrm{x}, n)$ iff $h(\mu(\mathrm{x})) = \langle l, \overline{w}_n \rangle$
3. $\theta(\mathrm{x}, l, \rho)$ iff $h(\mu(\mathrm{x})) = \langle l, \rho \rangle$
4. $\theta(\mathrm{x}, l)$ iff $h(\mu(\mathrm{x})) = \langle l, \rho \rangle$

*: x^* 는 x 가 새로운 힙 주소임을 의미한다.

그림 3.4 L-code의 축약 규칙

을 $\{\overline{w}\}$. ... 로 명시하고 있다. 본 논문에서 제시하는 STG-to-L-code 컴파일러는 그림 3.5와 같다.

$$
\begin{array}{ll}
CB[\![\{\overline{w}_m\}.\lambda\overline{x}_n.e]\!]\ \pi\ l & = \text{GETN } (\lambda z.\text{ARGCK } z\ (m+n)\ (\text{RET } z\ \lambda)\ (\text{GETC } z\ (\lambda(l,\overline{w}\,\overline{x}).CE[\![e]\!]\ (\text{GETNT}(\lambda z.\lambda t.\text{RET } z\ t)))))\\
CB[\![\{\overline{w}\}.c\ \overline{w}]\!]\ \pi\ l & = CN[\![\{\overline{w}\}.c\ \overline{w}]\!]\ l\\
CB[\![\{\overline{w}\}.e]\!]\ u\ l & = \text{GETN } (\lambda z.\text{GETC } z\ (\lambda(l,\overline{w}).CE[\![e]\!]\ (\text{GETNT}(\lambda z_r.\lambda t.\text{UPDC } z\ \langle l_{ind}, z_r\rangle)(\text{RET } z_r\ t)))))\\
CB[\![\{\overline{w}\}.e]\!]\ n\ l & = \text{GETN } (\lambda z.\text{GETC } z\ (\lambda(l,\overline{w}).CE[\![e]\!]\ (\text{GETNT}(\lambda z_r.\lambda t.\text{RET } z_r\ t))))\\
\\
CE[\![x_0\ \overline{x}_n]\!]\ \mathsf{L} & = \text{EVAL } x_0\ (\text{GETNT } (\lambda z_0.\lambda t_0.\text{APPLY } z_0\ x_1\ \ldots\ (\text{GETNT } (\lambda z_{n-1}.\lambda t_{n-1}.\text{APPLY } z_{n-1}\ x_n\ \mathsf{L}))\ldots)\\
CE[\![\text{case } e \text{ of } \{\overline{w}\}.\lambda x.alts]\!]\ \mathsf{L} & = \text{let } l_t = CA[\![\{\overline{w}\}.alt_t]\!]\ \mathsf{L}\ \text{in } CE[\![e]\!]\ (\text{GETNT } (\lambda x.\lambda t.\text{CASE } t\ \overline{l_t \rightarrow \langle l_t, w_0\,\overline{w}_t\rangle}))\\
CE[\![\text{let } \overline{x_i = \{\overline{w}_i\}.b_i}\ \text{in } e]\!]\ \mathsf{L} & = \text{let } \overline{l_i = CB[\![\{\overline{w}_i\}.b_i]\!]\ \pi_i\ l_i}\quad (b_i \not\equiv c\ \overline{y})\ \text{in PUTC } \overline{x_i \equiv clo_i}\ (CE[\![e]\!]\ \mathsf{L})\\
 & \quad \text{where}\quad clo_i \equiv \langle l_c, \overline{y}\rangle \quad \text{if } b_i \equiv c\ \overline{y}\\
 & \qquad\qquad clo_i \equiv \langle l_i, \overline{w}\rangle \quad \text{otherwise}\\
\\
CA[\![\{\overline{w}\}.c\ \overline{x} \rightarrow e]\!]\ \mathsf{L} & = \text{GETA } (\lambda(l,z\,\overline{w}).\text{GETC } z\ (\lambda(l_c,\overline{x}).CE[\![e]\!]\ \mathsf{L}))\\
CA[\![\{\overline{w}\}.default \rightarrow e]\!]\ \mathsf{L} & = \text{GETA } (\lambda(l,z\,\overline{w}).CE[\![e]\!]\ \mathsf{L})\\
\\
CN[\![\{w\}.ind\ w]\!]\ l & = \text{GETN } (\lambda z.\text{GETC } z\ (\lambda(l,w).\text{EVAL } w\ (\text{GETNT } (\lambda z.\lambda t.\text{RET } z\ t))))\\
CN[\![\{\overline{w}\}.c\ \overline{w}]\!]\ l & = \text{GETN } (\lambda z.\text{RET } z\ c)\\
\\
CP[\![\overline{decl}, e]\!] & = \text{let } l_{ind} = CN[\![\{w\}.ind\ w]\!]\ l_{ind}\\
 & \quad\ l_c = CN[\![\{\overline{w}_n\}.c\ \overline{w}_n]\!]\ l_c\quad (\top\ \overline{c\,\overline{n}} \in \overline{decl})\\
 & \text{in } CE[\![e]\!]\ (\text{GETNT } (\lambda z.\lambda t.\text{STOP } z))
\end{array}
$$

그림 3.5 STG에 대한 컴파일 규칙

이 컴파일러는 각각 바운드 식(bound expression)에 대한 *CB* 규칙, 식(expression)에 대한 *CE* 규칙, 선택자(alternative)에 대한 *CA* 규칙 그리고 보조 식(auxiliary expression)에 대한 *CN* 규칙이다. *CP* 규칙은 STG 프로그램을 입력으로 받아서 L-code 프로그램을 결과로 출력하며, 출력되는 L-code 프로그램은 후에 중첩된 단계의 닫힌 식을 한 단계의 닫힌 식으로 변환하는 호이스팅(hoisting) 변환을 하여 사용된다.

4. L-machine을 JVM으로 매핑

이 장에서는 STGM을 JVM으로 매핑 시 STGM의 전역 스택을 JVM으로 매핑하기 위한 방법과 그 문제점들을 살펴본다. 그리고 그 문제점들에 대한 해결방안으로서 3장에서 정의된 eval-apply 모델 기반의 STGM을 JVM으로 매핑하는 컴파일러를 제안한다.

4.1 STGM을 JVM으로 매핑할 때의 문제점

기본적으로 STGM의 하나의 클로저는 JVM에서 하나의 클래스로 매핑된다. 이 때 각 클로저들 사이에는 함수의 인자(argument)와 연속(continuation)을 건네기 위해 공유되는 전역적인 스택이 필요하다. 사실 STGM을 JVM으로 매핑하려는 하나의 동기로서 STGM과 JVM 모두가 공통적으로 스택 머신 아키텍처를 가진다는 점이 크게 작용하고있으나 실제로 이러한 STGM의 전역적 스택을 JVM으로 매핑하는 문제는 상당히 까다로운 문제로 나타나고 있다. 이 절에서는 STGM의 스택을 JVM으로 매핑하는 여러가지 가능성들과 문제점에 대해서 살펴본다. STGM의 스택을 JVM으로 매핑 하는 방법으로 생각해 볼 수 있는 것들은 다음과 같다.

4.1.1 STGM의 스택을 JVM의 JVM Stack으로 매핑

가장 직관적이고 자연스러운 방법이긴 하지만 클로저들 사이에 함수 인자들과 연속을 전달하기 위해서는 사용하는 스택에 대한 직접적인 제어(push, pop연산)가 필요하다. 그러나 JVM에서 제공하는 전역 스택인 JVM Stack은 프로그램에서 직접 제어할 수 있는 스택이 아니므로 클로저들 사이에 함수 인자와 연속을 전달하는 목적으로 사용하기에는 부적절하다.

4.1.2 STGM의 스택을 JVM의 로컬 스택(Local Stack)으로 매핑

각 프레임에 국부적으로 존재하는 로컬 스택(local stack)은 각 메쏘드들에 의해 지역적으로만 사용될 수 있는 스택이므로 모든 클로저들 사이에 공유되는 전역적인 용도로 사용하기에는 적합하지 않다. 하지만 모든 클로저를 하나의 메쏘드로 매핑한다면 이러한 방법이 가능성을 가지게 된다.

이 방법은 모든 클로저 코드들을 하나의 메쏘드 안에서 거대한 스위치 문장으로 묶고 스위치 문장의 각 케이스에 대해 하나의 클로저 코드를 매핑 함으로써 모든 클로저 사이의 이동이 그 스위치 문장을 포함하는 하나의 메쏘드 안에서 이루어지도록 하는 것이다. 이렇게 하면 메쏘드의 프레임에 할당되는 로컬 스택을 이용하여 모든 클로저들 사이의 인자와 연속을 전달하는 것이 가능하므로 전역적 스택의 효과를 얻을 수 있다.

그러나 자바 명세에서 클래스 파일에 대한 제약으로 로컬 스택의 크기와 메쏘드 코드의 크기를 각각 65K로 제약해 두고 있으므로[3] 이 방법은 컴파일할 프로그램이 커지면 사실상 사용 불가능한 방법이 된다.

4.1.3 전역적인 자료 구조를 이용한 STGM의 스택 구현

이 방법은 자바 언어에서의 배열(array)을 이용하여 STGM의 스택을 구현하는 방법이다. CLH[4]의 컴파일러에서도 이 방법을 사용하고 있으며 현재까지의 지연계산형 언어에 대한 자바 언어로의 매핑 방법에서 이용되는 방법이다. 이 방법에서는 전역적인 배열을 정적으로 할당하고 이것을 이용해서 클로저들간의 인자와 연속을 전달함으로써 STGM 스택의 역할을 구현하고 있다. 하지만 이 방법은 다음과 같은 문제점들을 가지고 있다[8].

- **배열 경계 검사**

현재 JVM에서는 모든 배열 연산에 대해 배열 접근시마다 배열의 경계에 대한 검사를 수행하게 된다. 이러한 검사는 STGM에서도 필요한 것이기는 하지만 JVM에서는 상당수의 경우 STGM에서는 불필요한 경계 검사를 수행하게 되는 부담이 발생한다.

• Pop연산 후의 널(null) 할당

자바에서는 기억 장소 재활용 체계를 사용하고 있으므로 메모리 유출(space leak)을 막기 위해서는 pop된 후 배열의 쓰이지 않는 부분에 대해서는 널(null)을 할당해 줘야 하는 부담이 발생한다.

• 정적인 배열 크기

배열로서 스택을 구현하게 됨으로써 스택의 크기가 정적으로 고정되는 결과를 낳게 된다. 그러므로 가상 기계가 다양한 크기의 프로그램 수행에 대해 유연하게 대응하지 못하는 문제가 발생한다.

• 기억 장소 재활용 체계 동작시의 부담

가상 기계가 큰 프로그램을 수행해야 할 경우 거대한 스택이 필요하게 되므로 배열의 크기도 커지게 된다. 이런 결과로 기억 장소 재활용 체계에서는 현재 프로그램에서 사용되는 스택의 크기에 상관없이 항상 전체의 거대한 배열에 대해 포인터 검사를 해야 하므로 상당한 부담으로 작용할 수 있다.

• STGM 아키텍처에 대한 수정

위와 같은 문제점들의 근본적 원인인 전역 스택에 대한 구현의 필요성은 전역 스택을 없앨 수 있다면 사라지게 된다. 그러므로 보다 적극적인 방법으로서 전역 스택을 사용하지 않는 아키텍처로의 수정을 생각해 볼 수 있다. 전역 스택의 용도는 다음의 두 가지로 나뉜다.

· 클로저들 사이의 인자를 전달

각 클로저들 사이에 인자를 전달하려면 모든 클로저에서 접근 가능한 매체가 있어야 한다. 현재 push-enter 모델의 STGM에서는 이를 위해 전역 스택을 사용하고 있다. 인자를 전달하기 위한 매체로서 스택 구조가 사용되어야 하는 이유는 push-enter 모델의 특징인 종단 호출(tail-call)에 기인한다. 즉, 종단 호출을 통해 클로저의 제어가 이동하는 경우에는 인자를 전달하는 시점과 사용하는 시점이 다를 수 있으므로 현재 전달된 인자가 사용되기 전에 또 다른 클로저로부터 인자가 전달될 수 있다. 따라서 이들을 전달하는 매체에는 임의의 개수의 인자들이 쌓일 수 있고 그 결과 크기가 동적으로 변할 수 있는 스택이 필요하게 된다.

하지만 만약 STGM을 eval-apply 모델에 기반하여 정의한다면 스택을 사용하지 않을 수 있다. 왜냐하면 eval-apply 모델에서는 인자를 전달하는 시점과 사용하는 시점이 항상 같기 때문이다. 즉, eval-apply 모델에서는 인자가 전달되는 즉시 인자를 받는 쪽에서 그 인자를 사용하므로 인자를 전달하는 매체에는 항상 고정된 개수의 인자만 저장되게 된다. 따라서 이러한 방식은 인자 전달을 위한 전역 스택의 필요성을 없앨 수 있다.

· 클로저들 사이의 연속을 전달

각 클로저들 사이의 연속 전달은 현재 STGM에서 채택하고 있는 push-enter 모델의 특징인 종단 호출(tail-call)에 기인하는 것이다. 그러므로 연속 전달은 STGM의 아키텍처를 종단 호출을 이용하지 않는 아키텍처로 수정함으로써 없앨 수 있다.

Push-enter 모델의 특징인 종단 호출은 호출된 클로저에 대한 연산이 끝난 후 프로그램의 제어가 호출한 클로저로 돌아가지 않고 또 다른 클로저로 이동하는 특징을 가지므로 클로저 호출시 호출된 클로저의 연산이 끝난 후 수행되어야 할 연속을 클로저 호출과 함께 넘겨줘야 한다. 그리고 이러한 연속 또한 임의의 개수가 쌓일 수 있으므로 push-enter 모델에서는 전역적인 스택이 필요하다.

하지만 만약 STGM이 eval-apply 모델에 기반하여 정의된다면 상황이 달라질 수 있다. 왜냐하면 eval-apply 모델의 경우에는 프로그램의 제어가 종단 호출을 이용하지 않고 클로저의 계산이 끝난 후 다시 호출한 클로저로 제어가 돌아오므로 JVM에서 제공하는 전통적인 함수 호출 메커니즘을 사용하는 것이 가능하기 때문이다. 그러므로 이러한 방식은 연속 전달을 위한 전역 스택의 필요성을 없앨 수 있다.

따라서 본 논문에서는 이러한 eval-apply 모델에 기반한 STGM을 제안하고 이를 JVM으로 매핑 하는 방법을 제안한다.

4.2 L-code-to-Java 컴파일러

이 절에서는 L-code로부터 자바 코드를 생성하는 컴파일러를 정의한다. 이를 위해 먼저 L-machine의 각 요소들을 자바 언어로 표현하기 위한 방법들을 제시하고 이에 기반하여 L-code-to-Java 컴파일러를 정의한다.

4.2.1 Java로의 표현

여기서는 자바 언어의 특징을 고려하여 L-machine의 각 요소들을 자바 언어로 효율적으로 표현하기 위한 방법들을 살펴본다.

• 클로저

각각의 클로저 *l*은 하나의 클래스로 표현된다. 그리고 모든 클로저 클래스들은 기저 클래스 *Clo*로부터 상속받도록 한다. *Clo* 클래스는 모든 클로저들이 가지는 공통적인 요소들을 멤버로하는 클래스로서 자기 자신을 가

리키는 공개 멤버 변수 *self*와 추상 멤버 함수 *code*()를 가진다.

```
public abstract class Clo {
    public Clo self = this;
    public abstract Clo code();
}
```

각 클로저에 대한 클래스는 그 클로저의 자유 변수를 멤버 변수로 두고 클로저에 대한 코드를 메쏘드로 둔다. 하지만 람다 식(λ-abstraction)에 대한 클로저를 표현할 때는 그 식의 인자에 대해서 해당하는 멤버 변수들과 받아들인 인자의 개수를 표시하는 멤버 변수를 추가해 준다. 이에 대한 표현은 아래와 같다.

```
public class Cl extends Clo {
    public Object f1,...,fn,a1,...,an;
    public int z;
    public Clo code() {...}
}
```

$f_1,...f_n$은 클로저의 자유 변수에 해당하고 $a_1,...,a_n$은 람다 식에 대한 인자에 해당하는 멤버 변수이다. 그리고 z는 현재까지 받아들인 인자의 개수를 나타내는 멤버 변수이다.

자유 변수와 인자에 대한 멤버 변수의 타입은 *Object* 타입으로 선언된다. 그리고 필요에 따라 제시된 컴파일 규칙에 의해 적합한 타입으로 전환(cast)된다.

• 실행 시간 시스템

Eval-apply 모델에서의 실행시간 시스템은 정적 변수(static variable) *node, tag, arg*를 포함한다. 다음은 이 컴파일러에서의 실행 시간 시스템을 나타내고 있다.

```
public class G {
    public static Object node;
    public static int tag;
    public static Object arg;
    public static void main(String[] args)
    throws CloneNotSupportedException {
        Cl r0 = new Cl();
        ((Clo) r0).code();
    }
}
```

*G.node*는 연산된 값이나 썽크에 대한 주소를, 그리고 *G.tag*는 태그를 전달하기 위해 사용되며 *G.arg*는 클로저들 사이에 인자를 전달하기 위한 매체로 사용된다. 그리고 C_l은 맨 처음에 수행되는 클로저이다.

레지스터 파일에 나타나는 매핑은 지역 변수 선언과 자바 언어의 가시 변수 규칙(scope rule)에 의해서 해결되며 스택과 힙은 JVM에 의해서 제공되는 스택과 힙으로 각각 표현된다.

• 갱신

기존의 기계에서 갱신은 주어진 주소에 새로운 값을 덮어서 씀으로써 이루어졌다. 하지만 JVM은 객체를 다른 객체 위에 덮어 쓰는 것을 허용하지 않으므로 갱신은 다른 방법을 통해 표현된다. 이를 위해 *Ind*라는 클래스를 도입한다.

```
public class Ind extends Clo {
  public Clo code() { return this.self; }
}
```

갱신이 이뤄질 수 있는 썽크(thunk)가 할당될 때 마다 *Ind* 클래스의 객체가 함께 생성되어 *self* 필드는 그 썽크를 가리키도록 한다. 후에 썽크가 계산되어 값으로 갱신될 때 *self* 필드가 그 계산된 값을 가리키도록 함으로써 갱신을 구현한다. *Ind* 클래스의 *code*()는 항상 *self* 필드가 가리키는 객체를 반환하도록 한다.

• 함수 적용

STGM에서는 함수에서 필요로 하는 인자 수 보다 전달된 인자 수가 적은 경우에 부분 적용식(partial application)을 생성한다.

본 논문에서 구현한 eval-apply 모델의 STGM에서는 인자에 대한 필드를 각 클로저 내부에 가지고 있으므로 새로운 부분 적용식을 생성하지 않고 원래 클로저의 필드를 채우는 것이 가능하다고 볼 수 있으나 그렇지 않다. 왜냐하면 생성된 부분 적용식을 또 다른 식에서 사용할 수가 있기 때문이다. 그러므로 본 논문의 구현에서는 함수 적용 시마다 부분 적용식을 생성하여 상태가 바뀐 클로저를 표현하도록 하였다.

그림 4.1에서 부분 적용식에 대한 클로저 생성은 자바의 *clone*() 메쏘드를 이용하였다. 이 때 L-code에 대한 축약 규칙에서는 **APPLY f x C** 명령어가 부분 적용식을 생성해서 힙에 할당하도록 하고 있지만 여기서는 **ARGCK** 명령어에서 부분 적용식을 생성하고 있다. 그 이유는 자바에서 *clone*() 메쏘드가 *protected*로 선언되어 있으므로 어떤 클래스를 복제(cloning)하는 것은 해당 클래스 자신과 그에 대한 상속 클래스에서만 가능하기 때문이다. 그러므로 **APPLY f x C** 명령어에 대한 자바 코드에서 **f** 클로저에 대한 클래스가 임의의 클래스일 수 있으므로 **APPLY f x**의 자바 코드가 속한 클래스에서는 **f** 클로저에 대한 복제가 불가능 할 수 있다. 따라서 **f** 클로저에 대한 복제는 **f**의 **ARGCK** 명령어에 대한 자바 코드에서 *this.clone*()으로 표현하여 자신의

클래스를 복제하도록 구현하고 있다.

• 제어 흐름

Eval-apply 모델에서는 함수를 계산되기 전의 환경과 함수가 인자에 적용될 때의 환경이 같다는 특징을 가진다. 이런 특징으로 인해 eval-apply 모델에서의 제어 흐름은 기존의 명령적 언어에서 함수 계산을 위해 사용하던 프레임 또는 활성 레코드(activation record)를 이용하는 계산 방식으로 표현할 수 있다[7]. 따라서 여기서는 JVM에서 제공하는 메쏘드 호출 메커니즘과 JVM Stack을 이용하여 제어 흐름을 표현한다.

4.2.2 컴파일러

여기서 정의하는 컴파일러는 eval-apply 모델에 기반한 L-code-to-Java 컴파일러이다. 이는 호이스팅(hoisting) 변환된 L-code 프로그램을 입력으로 하여 그림 4.1에서 제시한 컴파일 규칙에 따라 자바 코드를 생성한다.

이 컴파일 구도에서는 클로저를 하나의 클래스로 매핑하며 이 클래스는 클로저의 자유 변수와 매개 변수 그리고 현재까지 들어온 인자의 개수에 대한 변수를 멤버 변수로하고 클로저에 대한 코드를 메쏘드로 가지는 클래스이다.

컴파일러의 입력으로 들어오는 L-code 프로그램은 호이스팅(hoisting) 변환이 된 것이므로 각 **let** 바인딩은 내부에 또 다른 **let** 바인딩을 포함하지 않는다. 그러므로 각 클로저 레이블에 대하여 클래스 파일을 생성하는 것은 간단한 드라이버만으로 가능하다.

이 드라이버가 **let** 바인딩 $l=C$에 대해 **C**를 분석하여 자유 변수들을 구한 후 $l=\{w\}$, C와 같은 형태로 붙여주도록 함으로써 드라이버가 이들을 클래스의 멤버 변수를 만드는 정보로 이용할 수 있도록 한다. 그리고 클래스 내의 *code*() 함수에는 **C**에 대하여 컴파일 규칙에 의해 생성된 자바 코드가 삽입된다.

임의의 바운드 변수(bound variable)에 대하여 Object 타입의 지역 변수를 선언한다.

5. 실 험

본 논문에서는 CLH[4]의 논문에서 사용한 다섯 개의 Haskell 프로그램 fib 30, edigits 250, prime 500, soda, queen 8을 가지고 실험을 하였다. 실험 환경은 167MHz 프로세서와 256Mbytes메모리를 가지고 SUN OS 2.7을 운영 체제로 하는 SUN UltraSPARC 환경이다.

5.1 STG에서의 최적화

STGM을 사용함으로써 얻어지는 장점 가운데 하나는 GHC에서 수행하는 STG 레벨의 많은 최적화 연산들을

```
J[GETN (λx.C)]          = Object x = G.node; J[C]
J[GETNT (λx.λt.C)]      = Object x = G.node; int t = G.tag; J[C]
J[GETA (λ(l,x w).C)]    = J[C]
J[PUTC x_i ≡ ⟨l_i, w^j_{m_i}⟩_n C] = ... alloc_i; ... assign_j; ... J[C]
                          where alloc_i ≡ C_i x_i = new C_i()                  if π_i = n
                                        ≡ Ind x_i = new Ind()                  if π_i = u
                                          x_i.self = new C_i()
                                assign_j ≡ x_i.f_j = w^j_i                     if π_i = n
                                         ≡ ((C_i)x_i.self).f_j = w^j_i         if π_i = u
J[GETC y (λ(l,x_n).C)]  = Object x_1 = ((C_l)y).f_1; ...; x_n = ((C_l)y).f_n; J[C]
J[UPDC x ⟨l, w_n⟩ C]    = C_l o = new C_l(): o.f_1 = w_1; ... o.f_n = w_n;
                          ((C_l)x).self = o; J[C]
J[CASE t c_i → ⟨l_i, x_0 x_i⟩_n default → ⟨l_d, x_0 x_d⟩]
                        = switch(t) {
                            ... case c_i : { J[C_i] } ... default : { J[C_d] }
                          }
J[ARGCK x n C_1 C_2]     = if(G.arg != null) {
                            C_l y;
                            try {
                              y = this.clone();
                            }
                            catch (CloneNotSupportedException e) {
                              System.exit(0);
                            }
                            switch(f.z) {
                                case 0 : y.a_1 = G.arg; y.z = 1; break;
                                case 1 : y.a_2 = G.arg; y.z = 2; break;
                                 ...
                                case n-1 : y.a_n = G.arg; y.z = n; break;
                            }
                            G.arg = null;
                            if(n > y.z) { G.node = y; return;
                            } else { y.code(); return; }
                          } else
                            if(n > z) { J[C_1] } else { J[C_2] }
J[EVAL x C]             = G.node = x; x.code(); J[C]
J[APPLY f x C]          = G.arg = x; G.node = f; f.code(); J[C]
J[RET x t]              = G.node = x; G.tag = t; return;
J[STOP y]               = return;
```

그림 4.1 L-code에 대한 컴파일 규칙

이용할 수 있다는 점이다. 이상적으로는 본 논문의 컴파일러를 GHC의 후단(backend) 컴파일러로 생각할 수 있겠으나 여기서는 STG 언어와 같은 기능(operational behavior)을 가지는 STG와 유사한 형태의 함수 언어를 정의하였다. 본 논문에서 이용한 Haskell 벤치마크 프로그램에 대한 STG 프로그램은 GHC 4.04에서 최적화가 적용된 프로그램들로서 -ddump-stg 옵션을 사용하여 GHC로부터 STG 프로그램을 뽑아낸 후 약간의 수작업을 통해 본 논문에서 정의한 STG 언어로 변환해서 만들었다. 이러한 수작업을 통한 변환이 필요한 이유는 본 논문의 시스템에서는 GHC 라이브러리의 일부분만을 다루고 있기 때문이다.

5.2 결과

표 5.1과 5.2는 각 벤치마크 프로그램을 JDK 1.3과 JDK 1.2.2에서 수행한 결과를 각각 나타낸다. JDK 1.3에서는 본 논문에서 제시한 eval-apply 모델의 컴파일러가 push-enter 모델의 컴파일러보다 좋은 성능을 나타내고 있다. 하지만 JDK 1.2.2에서는 push-enter 모델이 eval-apply 모델보다 좋은 성능을 보이고 있다. 더우기 JDK 1.2.2에서는 전체적인 수행 시간이 JDK 1.3에서보다 빠르게 나타나고 있다. 한편 fib 프로그램은 JDK 1.2.2와 JDK 1.3 모두에서 eval-apply 모델이

push-enter 모델보다 빠르게 나타났다.

각 프로그램에 대한 수행 시간은 UNIX의 time 명령을 이용하여 측정하였다.

5.3 토의

표 5.1과 5.2에서 나타나듯이 JVM의 각 버전에 따라 eval-apply 모델과 push-enter 모델의 성능이 서로 상반되게 나타나고 있다. 이러한 현상은 각 버전의 JVM에서 배열 접근과 클로저 복제(cloning)에 대한 비용이 다르기 때문으로 보인다. 특이한 것은 fib 프로그램에 대해서는 항상 eval-apply 모델이 push-enter 모델보다 빠르게 나타나고 있다는 것이다. 여기서 fib 프로그램은 기초 연산(primmitive operation)들만 사용하는 프로그램으로서 기초 연산에 대해서는 GHC에서 부분 적용식이 발생하지 않도록 변환해 주므로 계산 모델에 관계 없이 fib에 대한 코드에서는 부분 적용이 발생하지 않는다.

한편 벤치마크 프로그램이 커질 경우 push-enter 모델의 스택에 대한 배열의 비용은 더욱 뚜렷이 드러날 것이다. 그 이유는 기억 장소 재활용 체계 동작 시마다 배열의 모든 원소에 대하여 검사를 해야 하는데 배열로 구현된 스택이 커질 경우 배열의 많은 원소에 대해서 모두 검사를 해야하는 부담이 나타나는 점과 큰 프로그램에서는 클로저들 사이의 인자와 연속 전달이 많아지므로 배열에 대한 접근 회수가 많아지게 되고 따라서 배열 접근에 대한 부담이 많아지는 점 때문이다.

표 5.1 JDK 1.3에서의 수행 시간

프로그램	Eval-apply	Push-enter
edigits	5.96s	6.86s
prime	5.66s	5.77s
queen	4.79s	5.9s
soda	3.11s	3.49s
fib	2.28s	6.47s

표 5.2 JDK 1.2.2에서의 수행 시간

프로그램	Eval-apply	Push-enter
edigits	4.21s	3.85s
prime	4.45s	3.63s
queen	4.01s	3.74s
soda	3.66s	3.26s
fib	2.51s	4.83s

6. 결론 및 향후 연구

본 논문에서는 기존의 push-enter모델의 STGM을 eval-apply 모델에 기반하여 새로이 정의하였다. 이것은 eval-apply 모델에 기반하면서도 여전히 STGM의 특징인 스파인 없음과 태그 없음의 성질을 그대로 살리므로 기존의 $<\nu, G>$-machine와는 다른 특징을 가진다. 그리고 이를 이용하여 STG로부터 자바로의 컴파일러를 정의함으로써 push-enter 모델에 기반한 컴파일러에서 야기되었던 문제점들을 해결할 수 있었다. 또한 이에 따른 실험 결과들을 제시함으로써 기존의 push-enter 모델에 기반한 컴파일러와의 비교가 이루어졌다.

향후 연구로는 클로저의 복제를 줄이기 위한 연구가 필요하다. 위에서 언급 했듯이 eval-apply 모델에서는 apply 연산시 마다 클로저를 복제해야 하는 부담이 존재한다. 그러므로 이를 해결하기 위해서는 정적 분석(static analysis)을 통하여 여러 개의 인자를 한꺼번에 넘기는 것이 가능한 경우를 분석해 내거나 두 개 이상의 인자를 동시에 넘기는 프로토콜의 개발이 필요하다.

또한 지금까지의 실험 결과는 작은 벤치마크 프로그램에 대한 결과이지만 보다 큰 실제 프로그램에 대해서 실험해 보는 것이 eval-apply 모델의 성능에 대한 더욱 정확한 판단에 도움이 될 것이다. 왜냐하면 현재와 같은 작은 프로그램만으로는 실제 큰 프로그램에서 스택 시뮬레이션이 끼치는 부담을 정당하게 평가할 수 없기 때문이다. 예를 들어 실제 큰 프로그램에 필요한 거대한 스택을 배열로 시뮬레이션 할 경우 기억장소 재활용 체계(garbage collector)가 동작할 때마다 현재 쓰이지 않는 스택의 많은 부분에 대한 검사가 발생하지만 작은 벤치마크 프로그램에서는 이러한 효과가 나타나지 않고 있다.

멀티 쓰레딩에 대한 연구도 가능하다. 멀티 쓰레딩을 할 경우 각 쓰레드마다 스택이 필요하게 된다. 만약 push-enter 모델을 사용하게 된다면 여러 개의 배열을 정적으로 할당해야 하므로 실제 쓰이지 않는 배열에 대한 메모리도 다른 쓰레드에서 사용할 수 없게 된다. 따라서 이는 쓰레드들의 동시 수행 기회를 줄이는 결과를 가져온다. 하지만 eval-apply 모델을 사용할 경우 JVM Stack을 이용하므로 각 쓰레드별로 그 시점에서 필요한 만큼의 스택만을 이용할 수 있고 결과적으로 더욱 많은 쓰레드들이 동시에 수행될 수 있는 이점을 가진다. 이러한 특징은 멀티 쓰레딩에 대한 연구로 이어질 수 있는 가능성을 제시한다.

그리고 현재 자바에서 제공하는 방대한 라이브러리를 효과적으로 사용하도록 하는 자바와의 혼합 프로그래밍에 대한 향후 연구 또한 필요하다.

참 고 문 헌

[1] S.L.P. Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, Vol 2, Part 2, pages 127-202, April 1992.

[2] T. Lindholm and F. Yellin. *The Java Virtual Machine™ Specification (2nd Ed.)*, Addison Wesley, 1999.

[3] J. Meyer and T. Downing. *JAVA Virtual Machine*, O'REILLY, 1997.

[4] K. Choi, H. Lim and T. Han. Compiling Lazy Functional Programs Based on the Spinless Tagless G-machine for the Java Virtual Machine. In *Fifth International Symposium on Functional and Logic Programming*, Waseda University, Tokyo, Japan, March 7-9, 2001. (Lecture Notes in Computer Science 2024).

[5] R. Dounce and P. Fradet. A Systematic Study of Functional Language Implementations. *ACM Transactions on Programming Languages and Systems*, Vol 20, No 2, pages 344-387, March 1998.

[6] M. Tullsen. Compiling Haskell to Java. 690 Project. September 1997.

[7] A. Vernet. The Jaskell Project. A diploma project, February 1999.

[8] D. Wakeling. Compiling Lazy Functional Programs for the Java Virtual Machine. *Journal of Functional Programming*, Vol 9, Part 6, pages 579-603, November 1999.

[9] D. Wakeling. A Haskell to Java Virtual Machine Code Compiler. In *Proceedings of the 1997 Workshop on the Implementation of Functional Languages*, pages 39-52, 1997.

남 병 규

1999년 2월 경북대학교 컴퓨터공학과 졸업. 2001년 2월 한국과학기술원 전산학과 졸업(공학석사.) 2000년 12월 ~ 현재 한국전자통신연구원 시스템연결망연구팀 재직 중. 관심분야는 컴퓨터구조, VLSI, 함수형 언어

최 광 훈

1994년 한국과학기술원 과학기술대학 전산학과 졸업. 1996년 한국과학기술원 전산학과 석사학위. 1996년 ~ 현재 한국과학기술원 전산학과 박사과정. 관심분야는 프로그래밍 언어, 컴파일러, 타입 시스템

한 태 숙

1976년 서울대학교 전자공학과 졸업. 1978년 한국과학기술원 전산학과 졸업. 1990년 Univ. of North Carolina at Chapel Hill 졸업. 현재 한국과학기술원 전자전산학과 부교수. 관심분야는 프로그래밍 언어론, 함수형 언어