중간언어 L-코드를 이용한 Haskell-Java 언어 번역기 구현

(Compiling Haskell to Java via an Intermediate Code L)

최 광 훈 † 한 태 숙 ††
(Kwanghoon Choi) (Taisook Han)

요 약 본 논문에서는 함수형 언어 Haskell을 Java언어로 변환하여 Java Virtual Machine에서 수행하는 컴파일 방법을 제안한다. 이 컴파일 방법은 추상 기계 Spineless Tagless G-machine (STGM)을 수행 모델로 삼는다. L-code로 명명한 중간 언어를 도입하여 추상기계의 각각의 세부 동작을 이 언어의 명령어로 표현하고, 일련의 세부 동작들을 이 언어의 바인딩을 통해 표현한다. 각 명령어는 Java의 문장으로 변환하기 쉽도록 정의하였다. Java에서의 표현 방법을 결정하고, STG 프로그램의 L-code 프로그램으로의 컴파일 규칙과 Java 프로그램으로의 컴파일 규칙을 제안한다. 실험을 통해 제안한 컴파일러에 의해 Haskell 프로그램으로부터 생성된 Java 프로그램의 수행 성능은 Glasgow Haskell 컴파일러의 최적화 변환을 적용했을 때 기존의 Haskell 인터프리터인 Hugs와 대등함을 보인다.

Abstract We propose a systematic method of compiling Haskell based on the Spineless Tagless G-machine (STGM) for the Java Virtual Machine (JVM). We introduce an intermediate language called L-code to identify each micro-operation of the machine by its instruction. Each macro operation of the machine is identified by a binding. Each instruction of the L-code can be easily translated into Java statements. After our determination on representation, an L-code program from a STG program is translated into a Java program according to our compilation rules. Our experiment shows that the execution times of translated benchmarks are competitive, compared with those in a Haskell interpreter Hugs, particularly when Glasgow Haskell compiler's STG-level optimizations are applied.

1. 서 론

자바가상기계를 목적 대상으로 하는 함수형 언어 컴파일러를 개발하고자 하는 동기는 다음의 두 가지 이유에서 비롯된다. 첫째, 자바가상기계를 포함하는 어떠한 컴퓨터 상에서도 이 코드를 실행시킬 수 있는 특징을 이용할 수 있고, 방대한 라이브러리가 이미 구축되어 있는 Java언어와 Haskell언어의 혼합 프로그래밍으로 함수형언어 사용을 촉진시킬 수 있다. 컴파일러 개발자 입장에서 볼 때, 함수형 언어 컴파일러에서 반드시 구현해야 할 메모리관리를 모든 자바가상기계에 이미 구현되어있는 메모리관리 방법으로 구현할 수 있는 용이한 측면이 있다. 또한 객체지향언어와 바이트코드 가상기계와 같은 환경에서 함수형 언어를 어떻게 구현하는가에 관한 흥미로운 연구가 될 수 있다.

본 논문의 목적은 자바가상기계를 목적 대상으로 하는 함수형 언어 Haskell 컴파일러를 개발하는 것이다. 이 컴파일러에서는 추상기계 STGM[1]을 실행 모델로 삼아 최종 코드를 만들고자 한다. STGM을 실행 모델로 채택한 이유는 Haskell과 같은 지연계산기반 함수형 언어의 실행 모델 중 가장 발전된 형태이기 때문이다. 그리고 STGM의 소스 언어인 STG에 대한 최적화 방법[2]이 다수 개발되어 있으므로 이를 적용하여 최종 코드의 실행 효율을 높일 수 있는 장점이 있다.

본 논문과 유사한 기존 연구로 Tullsen[3]에 의한 것과 Vernet[4]에 의한 것이 있다. 이들은 STG 프로그램을 직접 Java 프로그램으로 변환하고자 했다. 이들 연구의 문제점은 제한된 형태의 함수형 언어인 STG 자체로 실제 구현 수준의 사항들을 자세히 기술할 수 없다

† 비 회 원 : 한국과학기술원 전자전산학과
khchoi@pllab.kaist.ac.kr
†† 종신회원 : 한국과학기술원 전자전산학과 교수
han@cs.kaist.ac.kr
논문접수 : 2001년 3월 10일
심사완료 : 2001년 9월 15일

는 점에서 비롯된다. 즉, STG 언어와 Java 언어의 구조적인 차이 때문에 변환 과정이 복잡해져 이 과정을 정확히 기술하는 규칙 대신 서술적인 설명이나 예를 통해 설명할 수밖에 없었다. 결과적으로 이들 변환이 완전한지의 여부를 확신할 수 없다. 구체적으로 두 연구 모두 고차원 함수(higher-order function) 구현에 필수적인 테일 호출을 고려하지 않았다. 또한 이들 방법은 변환 방법자체와 STGM의 구성 요소를 Java로 표현하는 방법을 충분히 분리하지 않아 생성된 Java 클래스의 전체 크기를 줄일 수 있는 효율적인 표현 방법을 쉽게 적용할 수 없는 문제점을 지니고 있다.

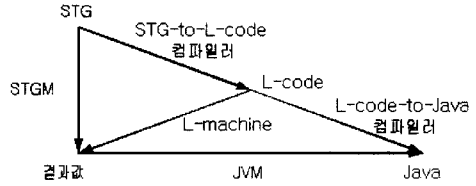

그림 1 전체 변환 과정

본 논문에서는 두 단계를 통한 변환 방법을 제시한다. 첫 번째 단계는 새로운 중간언어 L-code를 도입하여 STG 언어를 L-code로 변환하고, 두 번째 단계는 이 L-code를 Java 코드로 표현한다. STG 언어 자체에는 많은 내부 동작이 묵시적으로 내재되어 있으므로 L-code를 도입해 STGM 상에서 STG 프로그램의 동작을 명시적으로 기술하고자 도입하였다. L-code의 각 명령어는 STGM의 기본 요소들, 클로져, 레지스터 파일, 스택, 힙에 대한 연산을 구체적으로 기술하고 있고, 또한 이들 명령어들은 Java의 문장들로 쉽게 변환 가능하다. 그리고 L-code에서 L-code 명령어들로 이루어진 기본 블럭을 지정할 수 있다. STG 프로그램으로부터 유추된 기본 실행 단위인 클로져를 L-code에서 기본 블록으로 지정하고 궁극적으로 Java 클래스로 컴파일한다.

두 단계를 통한 변환 방법의 장점은 L-code를 통해 전체 변환 과정에 대한 정확한 규칙을 제시할 수 있고 이러한 구체적인 규칙을 제시함으로써 변환의 완전함에 대해 확신할 수 있다. 또한 전체 과정에서 변환 자체에 관한 과정과 Java로 표현하는 과정을 분리함으로써 각 단계에서의 이슈들을 독립적으로 다룰 수 있다.

본 논문에서는 제안한 두 단계 변환 방법에 기초한 컴파일러를 구현하여 5개의 벤치마크에 적용해서 생성한 Java 프로그램의 성능을 실험하였다.

본 논문의 구성은 다음과 같다. 2장에서 STG-to-L-code 컴파일러 규칙과 L-code 리덕션 기계를 통해 STGM을 체계적으로 구현할 수 있는 방법을 제시한다. 3장과 4장에서 L-code를 Java로 표현하는 방법을 결정하고 L-code-to-Java 컴파일러 규칙을 제시한다. 5장에서 생성된 Java 프로그램의 성능을 실험한다. 6장에서 관련 연구들과 비교를 하고 7장에서 결론을 맺는다.

2. Spineless Tagless G-machine (STGM)

이 장에서는 추상기계 STGM을 STG, L-code, L-machine, STG-to-L-code 컴파일러를 통해 정의한다. L-machine와 STG-to-L-code 컴파일러는 STGM을 정의하는 명세이다.

2.1 입력 언어 : STG

STGM의 입력 언어인 STG의 구문은 다음과 같다.

x, y, z, w			변수
v	::=	$\lambda\overline{x}.e \mid c\ \overline{x}$	값
e	::=	$x_0\ \overline{x} \mid \text{let } \overline{bind} \text{ in } e \mid \text{case } e \text{ of } \lambda z.alts$	식
b	::=	$v \mid e$	바운드식
π	::=	$u \mid n$	갱신표시
$bind$	::=	$x \overset{\pi}{=} b$	바인딩
t	::=	$c \mid \lambda \mid \text{default}$	태그
$alts$	::=	$\overline{alt}$	선택자그룹
alt	::=	$c\ \overline{x} \rightarrow e \mid \text{default} \rightarrow e$	선택자
$decl$	::=	$\mathrm{T}\ \overline{c_i\ n_i}$	타입선언
prg	::=	$\overline{decl}, \text{let } \overline{bind} \text{ in } main$	프로그램

구문 정의에서 사용한 표기법 $\overline{\text{obj}}_n$은 $\text{obj}_1 \ldots \text{obj}_n$ ($n \geq 0$)을 나타낸다. 구문 정의에서 다루는 식으로 정규형(weak head normal form) v와 비정규형인 e가 있다. 바인딩에서 π는 갱신 표시기로, u는 b를 계산하여 얻은 결과 값을 x에 갱신해야함을 지정하고 n은 갱신하지 않음을 지정한다. case e of $\lambda z.alts$에서 e를 계산하여 얻은 값이 z에 바인딩된다. 값의 종류를 구분할 목적으로 태그를 사용한다. 데이타 컨스트럭터(data constructor)에 대해 태그 c를, 람다식(lambda expression)에 대해 태그 λ를 사용한다. 편의상 default도 태그로 간주한다. STG 프로그램은 하나의 식 e와 데이타 컨스트럭터의 태그와 인자 개수를 정의하는 타입 선언문으로 이뤄진다. 구문에 관한 자세한 내용은 Peyton Jones의 논문 [1]에 나와있다. 다음은 STG 코드의 간단한 예이다.

$$\text{let}\quad s \overset{n}{=} \lambda f\ g\ x.\ \text{let}\ a \overset{u}{=} g\ x\ \text{in}\ f\ x\ a\quad \text{in} \ldots$$

2.2 중간 언어 : L-code

STG 프로그램을 Java 프로그램으로 변환하기 위한 중간 단계로 L-code 언어를 도입한다. 이 언어의 구문은 다음과 같다.

```
x,y,z,w,t,s                                                          레지스터
l                                                                    라벨
C ::= let l̄ = C̄ in C                                                 L코드
    | JMPC x          | JMPK x y t          | CASE t t̄ → ⟨l, x0 x̄⟩
    | GETN (λx.C)     | GETNT (λx.λt.C)     | GETA (λ⟨l, x0 x̄⟩.C)
    | PUTARG x̄ C      | GETARG (λx̄.C)       | ARGCK n Cmp C
    | PUTK ⟨l, x̄⟩ C   | GETK (λ⟨l, x̄⟩.C)    | REFK (λx.C)
    | SAVE (λs.C)     | DUMP C              | RESTORE s C
    | PUTC x ≝ ⟨l, x̄⟩ C | GETC x (λ⟨l, w̄⟩.C) | UPDC x ⟨l, w̄⟩
    | PUTP ⟨l, x̄⟩ (λy.C) | STOP x
```

L-code에서 사용되는 변수를 레지스터(register)로 부른다. L-code 명령어들의 시작 주소를 지정할 목적으로 라벨을 사용한다. L-code의 let블록의 바인딩에서 각 라벨과 연관된 L-code명령어를 정의한다. STG 프로그램은 궁극적으로 다단계 let블록 형태의 L-code 프로그램으로 변환된다. 각각의 L-code 명령어 의미는 다음 장에서 정의할 환원 규칙(reduction rule)으로 정의한다.

L-code를 도입한 목적은 STGM을 구성하는 모든 종류의 단위 동작을 각각 하나의 L-code 명령어로 지정하고, 이 단위 동작으로 이뤄진 좀 더 큰 실행 단위인 클로져(closure)를 let 블록의 바인딩으로 지정하는 것이다. L-code로 STG 프로그램의 동작을 완전하게 제시할 수 있고, 이를 통해서 Java 프로그램으로의 변환과정을 정확하게 구현할 수 있다. 특히 Java 언어가 객체지향 언어(object-oriented language)임에도 불구하고 L-code에 이 패러다임을 위한 별도의 특징을 도입하지 않고도 함수형언어의 클로져를 객체지향형언어의 오브젝트로 간주하는 원칙(*closures as objects*)하에 Java 프로그램으로 간단 명료하게 변환할 수 있다.

2.3 L-machine

L-machine은 [그림 2]에서 제시한 환원 규칙으로 정의한다. L-machine은 실행시간에 다음의 자료구조를 사용한다.

```
⟨l, ρ⟩                                                               클로져
ρ ::= x̄                                                              환경
μ ::= μ0 | μ[x ↦ x] | μ[x ↦ l] | μ[t ↦ t] | μ[s ↦ s]                레지스터파일
s ::= s0 | x s | ⟨l, ρ⟩ s                                           스택
h ::= h0 | h[x ↦ ⟨l, ρ⟩] | h[l ↦ C]                                  힙
```

L-machine은 주어진 상태의 패턴에 따라 환원 규칙을 선택해 적용한다. L-machine의 상태는 주로 $C\ \mu\ s\ h$ 형태이다. 나머지 형태는 L-code 명령과 함께 설명한다. 레지스터 파일 μ는 주어진 레지스터에 이와 연관된 힙 주소, 라벨, 태그, 스택을 내는 함수이다. 힙 주소는 STG의 변수와 같은 구문으로 다룬다. 환경 ρ는 힙 주소의 벡터 형태이다. 클로져 $\langle l, \rho\rangle$는 코드 주

(**let** $\overline{l_i = C_i}$ **in** C) $\mu\ s\ h$	$\Rightarrow$	C $\mu\ s\ h[\overline{l_i \mapsto C_i}]$	
(STOP x) $\mu\ s\ h$	$\Rightarrow$	$(\mu(x), h)$	
(JMPC x) $\mu\ s\ h$	$\Rightarrow$	$h(l)\ \mu(x)\ s\ h$	$\theta(x, l, \rho)$[1]
(GETN (λx.C)) $x\ s\ h$	$\Rightarrow$	C $\mu_0[x \mapsto x]\ s\ h$	
(JMPK y x t) $\mu\ s\ h$	$\Rightarrow$	$h(\mu(y))\ \mu(x)\ \mu(t)\ s\ h$	
(GETNT $(\lambda x.\lambda t.C)\ x\ t\ s\ h$	$\Rightarrow$	C $\mu_0[x \mapsto x, t \mapsto t]\ s\ h$	
(CASE t $t_i \rightarrow \langle l_i, x_0\ \overline{x_i}\rangle$) $\mu\ s\ h$	$\Rightarrow$	$h(l_j)\ \langle l_j, \mu(x_0)\ \overline{\mu(x)}_j\rangle\ s\ h$	if $\exists j.\mu(t) = t_j$
	$\Rightarrow$	$h(l_d)\ \langle l_d, \mu(x_0)\ \overline{\mu(x)}_d\rangle\ s\ h$	otherwise[2]
(GETA $\lambda\langle l, x_0\ \overline{x}\rangle$.C) $\langle l, x_0\ \overline{x}\rangle\ s\ h$	$\Rightarrow$	C $\mu_0[x_0 \mapsto x_0, \overline{x \mapsto x}]\ s\ h$	
(GETARG $(\lambda\overline{x}_n.C)$) $\mu\ (\overline{x}_n\ s)\ h$	$\Rightarrow$	C $\mu[\overline{x \mapsto x}]\ s\ h$	
(PUTARG $\overline{x}$ C) $\mu\ s\ h$	$\Rightarrow$	C $\mu\ (\overline{\mu(x)}\ s)\ h$	
(ARGCK $n\ C_{mp}$ C) $\mu\ (\overline{x}_m\ \langle l, \rho\rangle\ s)\ h$	$\Rightarrow$	C $\mu\ (\overline{x}_m\ \langle l, \rho\rangle\ s)\ h$	if $m \geq n$
	$\Rightarrow$	$C_{mp}\ \mu\ (\overline{x}_m\ \langle l, \rho\rangle\ s)\ h$	otherwise
(GETK $(\lambda\langle l, \overline{x}\rangle.C)$) $\mu\ (\langle l, \overline{x}\rangle\ s)\ h$	$\Rightarrow$	C $\mu[\overline{x \mapsto x}]\ s\ h$	
(PUTK $\langle l, \overline{x}\rangle$ C) $\mu\ s\ h$	$\Rightarrow$	C $\mu\ (\langle l, \overline{\mu(x)}\rangle\ s)\ h$	
(REFK (λx.C)) $\mu\ (\langle l, \rho\rangle\ s)\ h$	$\Rightarrow$	C $\mu[x \mapsto l]\ (\langle l, \rho\rangle\ s)\ h$	
(SAVE (λs.C)) $\mu\ s\ h$	$\Rightarrow$	C $\mu[s \mapsto s]\ s\ h$	
(DUMP C) $\mu\ (\langle l, \rho\rangle\ s)\ h$	$\Rightarrow$	C $\mu\ \langle l, \rho\rangle\ h$	
(RESTORE s C) $\mu\ s\ h$	$\Rightarrow$	C $\mu\ \mu(s)\ h$	
(GETC x $(\lambda\langle l, \overline{w}\rangle.C)$) $\mu\ s\ h$	$\Rightarrow$	C $\mu[\overline{w \mapsto w}]\ s\ h$	$\theta(x, l, \overline{w})$
(PUTC $x \stackrel{n}{=} \langle l, \overline{w}\rangle$ C) $\mu\ s\ h$	$\Rightarrow$	C $\mu'\ s\ h[x^* \mapsto \langle l, \overline{\mu'(w)}\rangle]$	$\mu' = \mu[\overline{x \mapsto x^*}]$[3]
(UPDC x $\langle l, \overline{w}\rangle$ C) $\mu\ s\ h$	$\Rightarrow$	C $\mu\ s\ h[\mu(x) \mapsto \langle l, \overline{\mu(w)}\rangle]$	
(PUTP $\overline{\langle l_i, x_0\rangle}_n$ $(\lambda y.C)$) $\mu\ (\overline{x}_m\ \langle l', \rho\rangle\ s)\ h$			$n > m$
$\Rightarrow$ C $\mu[y \mapsto y^*]\ (\langle l', \rho\rangle\ s)\ h[y^* \mapsto \langle l_{m+1}, \mu(x_0)\ \overline{x}_m\rangle]$			

1. $\theta(x, l, \rho)$는 주어진 h와 μ에 대해서 $h(\mu(x)) = \langle l, \rho\rangle$로 정의
2. default $\rightarrow \langle l_d, x_0\ \overline{x}_d\rangle$를 선택
3. x^*는 새로 할당된 힙주소

그림 2 L-machine의 환원 규칙

소로 주어진 라벨 l과 힙 주소 벡터인 환경 ρ로 구성된다. 스택 s의 원소는 힙 주소나 클로져이다. 힙 h는 힙 주소를 받아 그 주소의 클로져를 내고, 라벨을 받아 그 라벨의 코드를 내는 함수이다. μ_0, s_0, h_0는 각각 초기 레지스터 파일, 초기 스택, 초기 힙을 나타낸다.

각 L-code 명령어에 대해 간단히 설명하면 다음과 같다. JMPC x는 레지스터 x에 저장된 힙 주소에 놓인 클로져의 코드를 실행한다. JMPK y x t는 레지스터 y에 저장된 라벨에 연결된 코드에 레지스터 x에 저장된 클로져 주소와 레지스터 t에 저장된 태그를 전달하면서 그 코드를 실행한다. CASE t $\overline{t_i \cdot \langle l_i, x_0, \overline{x_i}\rangle}$는 레지스터 t에 저장된 태그에 따라 실행할 코드 l_i를 결정하고 그 코드에 레지스터 x_0와 $\overline{x_i}$에 저장된 주소를 전달하면서 실행한다. GETN, GETNT, GETA는 각각 JMPC, JMPK, CASE에서 전달한 주소 혹은 태그를 받는다. GETN을 실행하는 상태에서는 레지스터 파일 대신 힙 주소가, GETNT를 실행하는 상태에서는 레지스터 파일 대신 힙 주소와 태그가, GETA를 실행하는 상태에서는 레지스터 파일 대신 클로져가 나타난다. PUTARG는 주어진 레지스터에 저장된 주소를 스택에 쌓고, GETARG는 스택에 저장된 주소를 지정된 레지스터로 옮긴다. ARGCHK n C_{mp} C는 주어진 스택에 인자가 n개 있는지를 검사하고 만일 그렇다면 C를 실행하고 그렇지않다면 C_{mp}를 실행한다. PUTK, GETK, REFK는 스택에 저장될 클로져를 다룬다. PUTC, GETC, UPDC, PUTP는 힙에 저장될 클로져를 다룬다. SAVE, DUMP, RESTORE는 스택을 보관하고 없애고 다시 복구한다. STOP은 L-machine의 수행을 멈추고 주어진 레지스터의 주소를 L-code 프로그램 수행 결과값으로 낸다. 다음은 L-code 명령어에 환원규칙을 적용한 실행 과정의 예이다.

(GETN (λz.GETC z (λ⟨l_s,⟩.ARGCK 3 C_{mp} C_r))) z (f g x s_0) h
⇒ (GETC z (λ⟨l_s,⟩.ARGCK 3 C_{mp} C_r)) $\mu_0[z \mapsto z]$ (f g x s_0) h
where $h(z) = \langle l_s, \rangle$
⇒ (ARGCK 3 C_{mp} C_r) $\mu_0[z \mapsto z]$ (f g x s_0) h
where $C_r \equiv$ GETARG (λf g x.(**let** l_a = C_a**in** C_1))
⇒ (GETARG (λf g x.(**let** l_a = C_a**in** C_1))) $[z \mapsto z]$ (f g x s_0) h
⇒ (**let** l_a = C_a**in** C_1) $\mu_0[z \mapsto z, f \mapsto f, g \mapsto g, x \mapsto x]$ s_0 h
where $C_1 \equiv$ PUTC a ≝⟨l_a, g x⟩ (PUTARG x a C_2), $C_2 \equiv$ JMPC f
⇒ (PUTC a ≝⟨l_a, g x⟩ (PUTARG x a C_2)) $\mu_0[z \mapsto z, \ldots, x \mapsto x]$ s_0 $h[l_a \mapsto C_a]$
⇒ (PUTARG x a C_2) $\mu_0[z \mapsto z, \ldots, x \mapsto x, a \mapsto a]$ s_0 $h[l_a \mapsto C_a, a \mapsto \langle l_a, g\, x\rangle]$
where $a \notin h[l_a \mapsto C_a]$
⇒ (JMPC f) $\mu_0[z \mapsto z, \ldots, x \mapsto x, a \mapsto a]$ (x a s_0) $h[l_a \mapsto C_a, a \mapsto \langle l_a, g\, x\rangle]$

궁극적으로 L-machine은 주어진 L-code 프로그램의 결과로 특정 힙 클로져 주소와 마지막 상태의 힙을 낸다.

2.4 STG 프로그램을 L-code로 컴파일 하는 방법

우선 STG 구문에 몇 가지 보조 구문을 추가한다. 부분 적용 *pap* w_0 $\overline{w_n}$, 간접 참조 ind w, 표준 데이타 컨스트럭터 c $\overline{w_n}$, 선택 컨티뉴에이션 *sel* $\lambda z.alts$, 갱신 컨티뉴에이션 *upd* w_1 w_2, 정지 컨티뉴에이션 *halt*를 추가한다.

[그림 3]에서 STG 컴파일러 규칙을 제시한다. 편의상 몇몇 컴파일러 규칙에서 $\{\overline{w}\}.\lambda\overline{x}.e$에서처럼 STG 구문에 포함된 자유변수의 집합을 나타내는 주석 $\{\overline{w}\}$을 사용한다. 컴파일 규칙에서 심볼 테이블 τ를 사용한다. 심볼 테이블은 STG 구문의 변수 이름을 레지스터 이름으로 변환한다. τ_0는 초기 심볼 테이블이다.

$\tau ::= \tau_0 \mid \tau[x \mapsto x]$ 심볼테이블

컴파일러 규칙은 다섯 가지 종류로 분류할 수 있다. let에서 정의된 식에 대한 컴파일 규칙 B, 단순 식에 대한 컴파일 규칙 E, 선택자 그룹에 대한 규칙 A, 보조 구문으로 도입한 여러 컨티뉴에이션에 대한 규칙 C, 이외에 보조 컴파일 규칙 X가 있다. P 규칙은 STG 프로그램을 받아 여러 레벨의 let 블록으로 구성된 L-code 프로그램을 낸다.

[그림 3]에서 제시한 컴파일 규칙에 의하면 let에서 정의된 L-code 명령어 C는 자유 변수를 갖지 않는다. 코드와 데이타를 분리하는 과정을 클로져 변환이라 하는데 이 컴파일 규칙은 STG 프로그램에 대한 클로져 변환이라 할 수 있다. 자유 변수를 갖지 않는 명령어들로만 이루어진 바인딩은 최상위 레벨의 let 블록으로 옮길 수 있다. 이런 방식으로 모든 정의를 단 하나의 최상위 레벨의 let 블록에 모든 바인딩이 위치하도록 변환이 가능하다. 이를 호이스팅(hoisting) 변환이라 한다. 다음은 STG 프로그램을 컴파일한 예제이다.

let l_{upd} = ... **in**
let l_s = $B[\{\}.\lambda f\ g\ x.$ **let** $a \stackrel{=}{} g\ x$ **in** $f\ x\ a]$ n l_s **in** ...
= **let** l_{upd} = ... **in**
let l_s = GETN (λz.GETC z (λ⟨l_s,⟩.ARGCK 3 (...)(GETARG (λf g x.
let l_a = GETN (λz.GETC z (λ⟨l_a, g x⟩.SAVE (λs.PUTK ⟨l_{upd}, z s⟩
(DUMP (PUTARG x (JMPC g))))))
in PUTC a ≝⟨l_a, g x⟩ (PUTARG x a (JMPC f))))))
in ...

다음은 컴파일 과정으로 얻어진 L-code에 호이스팅 변환을 적용하여 모든 정의가 최상위 let 블록으로 옮겨진 형태의 L-code이다. 이 형태가 STG-to-L-code의 최종 결과로 다음 단계의 L-code 컴파일러 입력이 된다.

let l_{upd} = ...
l_s = GETN (λz.GETC z (λ⟨l_s,⟩.ARGCK 3 (...)(GETARG (λf g x.
PUTC a ≝⟨l_a, g x⟩ (PUTARG x a (JMPC f))))))
l_a = GETN (λz.GETC z (λ⟨l_a, g x⟩.(SAVE (λs.PUTK ⟨l_{upd}, z s⟩
(DUMP (PUTARG x (JMPC g)))))))
in ...

주어진 STG 프로그램을 컴파일 하여 얻은 L-code

$$
\begin{array}{ll}
B[\![\{\overline{w}\}.\lambda\overline{x}_n.e]\!]\ \pi\ l & = \mathsf{GETN}\ (\lambda \mathsf{z}.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l, \overline{\mathsf{w}}\rangle.\mathsf{ARGCK}\ n \\
 & \quad (\mathsf{PUTP}\ \overline{\langle l_{pap,i}, \mathsf{z}\rangle}_n\ (\lambda \mathsf{p}.\mathsf{REFK}\ (\lambda \mathsf{y}.\mathsf{JMPK}\ \mathsf{y}\ \mathsf{p}\ \lambda))) \\
 & \quad (\mathsf{GETARG}\ (\lambda \overline{\mathsf{x}}.E[\![e]\!]\ \tau_0[\overline{w \mapsto \mathsf{w}}, \overline{x \mapsto \mathsf{x}}]))))) \\
B[\![\{\overline{w}\}.c\ \overline{w}]\!]\ \pi\ l & = X[\![\{\overline{w}\}.c\ \overline{w}]\!]\ l \\
B[\![\{\overline{w}\}.e]\!]\ u\ l & = \mathsf{GETN}\ (\lambda \mathsf{z}.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l, \overline{\mathsf{w}}\rangle.\mathsf{SAVE}\ (\lambda \mathsf{s}.\mathsf{PUTK}\ \langle l_{upd}, \mathsf{z}\,\mathsf{s}\rangle \\
 & \quad (\mathsf{DUMP}\ (E[\![e]\!]\ \tau_0[\overline{w \mapsto \mathsf{w}}])))))) \\
B[\![\{\overline{w}\}.e]\!]\ n\ l & = \mathsf{GETN}\ (\lambda \mathsf{z}.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l, \overline{\mathsf{w}}\rangle.E[\![e]\!]\ \tau_0[\overline{w \mapsto \mathsf{w}}])) \\
 & \\
E[\![x_0\ \overline{x}]\!]\ \tau & = \mathsf{PUTARG}\ \overline{\tau(x)}\ (\mathsf{JMPC}\ \tau(x_0)) \\
E[\![\mathsf{case}\ e\ \mathsf{of}\ \{\overline{w}\}.\lambda x.alts]\!]\ \tau & = \mathbf{let}\ l_{sel} = C[\![\{\overline{w}\}.sel\ \lambda x.alts]\!]\ l_{sel} \\
 & \quad \mathbf{in}\ \mathsf{PUTK}\ \langle l_{sel}, \overline{\tau(w)}\rangle\ (E[\![e]\!]\ \tau) \\
 & \quad \mathrm{where}\ l_{sel}\ \mathrm{fresh} \\
E[\![\mathrm{let}\ \overline{x_i \overset{\pi_i}{=} \{\overline{w}_i\}.b_i}\ \mathrm{in}\ e]\!]\ \tau & = \mathbf{let}\ \overline{l_i = B[\![\{\overline{w}_i\}.b_i]\!]\ \pi_i\ l_i}\quad (b_i \not\equiv c\ \overline{y}) \\
 & \quad \mathbf{in}\ \mathsf{PUTC}\ \overline{\mathsf{x}_i \overset{\pi_i}{=} \langle l_i, \overline{\mathsf{w}}_i\rangle}\ (E[\![e]\!]\ \tau[\overline{x_i \mapsto \mathsf{x}_i}]) \\
 & \quad \mathrm{where}\quad l_i \equiv l_c\quad \mathrm{and}\ \mathsf{w}_i \equiv \tau[\overline{x_i \mapsto \mathsf{x}_i}](y)\qquad \mathrm{if}\ b_i \equiv c\ \overline{y} \\
 & \qquad\qquad l_i\ \mathrm{fresh\ and}\ \mathsf{w}_i \equiv \tau[\overline{x_i \mapsto \mathsf{x}_i}](w)\qquad \mathrm{otherwise} \\
 & \\
A[\![\{\overline{w}\}.c\ \overline{x} \to e]\!]\ l & = \mathsf{GETA}\ (\lambda\langle l, \mathsf{z}\,\overline{\mathsf{w}}\rangle.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l_c, \overline{\mathsf{x}}\rangle.E[\![e]\!]\ \tau_0[\overline{w \mapsto \mathsf{w}}, \overline{x \mapsto \mathsf{x}}])) \\
A[\![\{\overline{w}\}.\mathrm{default} \to e]\!]\ l & = \mathsf{GETA}\ (\lambda\langle l, \mathsf{z}\,\overline{\mathsf{w}}\rangle.E[\![e]\!]\ \tau_0[\overline{w \mapsto \mathsf{w}}]) \\
 & \\
C[\![\{w_1, w_2\}.upd\ w_1\ w_2]\!]\ l & = X[\![\{w_1, w_2\}.upd\ w_1\ w_2]\!]\ l \\
C[\![\{\overline{w}\}.sel\ \lambda w_0.\overline{alt_i}]\!]\ l & = \mathbf{let}\ \overline{l_i = A[\![alt_i]\!]\ l_i} \\
 & \quad \mathbf{in}\ \mathsf{GETNT}\ (\lambda \mathsf{w}_0.\lambda \mathsf{t}.\mathsf{GETK}\ (\lambda\langle l, \overline{\mathsf{w}}\rangle.\mathsf{CASE}\ \mathsf{t}\ \overline{t_i \to \langle l_i, \mathsf{w}_0\ \overline{\mathsf{w}}_i\rangle})) \\
 & \quad \mathrm{where}\quad \overline{l_i}\ \mathrm{fresh},\ alt_i \equiv \{\overline{w}_i\}.t_i \ldots \to \ldots \\
C[\![\{\}.halt]\!]\ l & = X[\![\{\}.halt]\!]\ l \\
 & \\
X[\![\{w_0, \overline{w}\}.pap\ w_0\ \overline{w}]\!]\ l & = \mathsf{GETN}\ (\lambda \mathsf{z}.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l, \mathsf{w}_0\,\overline{\mathsf{w}}\rangle.\mathsf{PUTARG}\ \overline{\mathsf{w}}\ (\mathsf{JMPC}\ \mathsf{w}_0))) \\
X[\![\{w\}.ind\ w]\!]\ l & = \mathsf{GETN}\ (\lambda \mathsf{z}.\mathsf{GETC}\ \mathsf{z}\ (\lambda\langle l, \mathsf{w}\rangle.\mathsf{JMPC}\ \mathsf{w})) \\
X[\![\{\overline{w}\}.c\ \overline{w}]\!]\ l & = \mathsf{GETN}\ (\lambda \mathsf{z}.\mathsf{REFK}\ (\lambda \mathsf{x}.\mathsf{JMPK}\ \mathsf{x}\ \mathsf{z}\ c)) \\
X[\![\{w_1, w_2\}.upd\ w_1\ w_2]\!]\ l & = \mathsf{GETNT}\ (\lambda \mathsf{z}.\lambda \mathsf{t}.\mathsf{GETK}\ (\lambda\langle l, \mathsf{w}_1\,\mathsf{w}_2\rangle. \\
 & \quad\quad \mathsf{RESTORE}\ \mathsf{w}_2\ (\mathsf{UPDC}\ \mathsf{w}_1\ \langle l_{ind}, \mathsf{z}\rangle\ (\mathsf{JMPC}\ \mathsf{z})))) \\
X[\![\{\}.halt]\!]\ l & = \mathsf{GETNT}\ (\lambda \mathsf{z}.\lambda \mathsf{t}.\mathsf{GETK}\ (\lambda\langle l, \rangle.\mathsf{STOP}\ \mathsf{z})) \\
 & \\
P[\![\overline{decl}, e]\!] & = \mathbf{let}\ l_{pap,m} = X[\![\{\overline{w}_m\}.pap\ \overline{w}_m]\!]\ l_{pap,m}\quad (m \geq 1) \\
 & \quad\quad l_{ind} = X[\![\{w\}.ind\ w]\!]\ l_{ind} \\
 & \quad\quad l_{upd} = X[\![\{w_1, w_2\}.upd\ w_1\ w_2]\!]\ l_{upd} \\
 & \quad\quad l_{halt} = X[\![\{\}.halt]\!]\ l_{halt} \\
 & \quad\quad l_c = X[\![\{\overline{w}_n\}.c\ \overline{w}_n]\!]\ l_c\quad (\mathrm{T}\ \overline{c\ n} \in \overline{decl}) \\
 & \quad \mathbf{in}\ \mathsf{PUTK}\ \langle l_{halt}, \rangle\ (E[\![e]\!]\ \tau_0)
\end{array}
$$

그림 3 STG 컴파일러의 컴파일 규칙

프로그램을 C라 하면 L-machine은 $C\ \mu_0\ s_0\ h_0$를 초기 상태로 앞에서 정의한 환원 규칙을 적용하여 L-code 프로그램을 실행한다.

3. 표현 방법(Representation)

이 장부터 L-code 프로그램을 Java 프로그램으로 변환하는 방법에 대해 설명한다. 우선 L-code 컴파일러를 제시하기 전에, L-code와 L-machine을 Java로 변환하는 동안 이 컴파일러에서 염두하고 있는 표현 방법들을 설명한다. 다음 장에서 이 표현 방법에 기반해서 L-code 컴파일러를 제시한다.

3.1 클로져

모든 클로져의 공통 요소를 추상 클래스(abstract class) Clo로 표현한다. 이 클래스는 멤버 변수 ind와 추상 멤버 함수 code()를 포함한다. 멤버 변수 ind는 이 클래스의 객체가 만들어지는 시점에 자기 자신을 지칭하는 this로 초기화된다.

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

주어진 클로져를 표현하는 클래스는 추상 클래스 Clo의 하위 클래스로 정의한다. 새로 정의하는 클래스에는 클로져의 환경에 포함된 자유변수에 대한 필드와 클로져의 코드에 대한 Java 문장이 위치할 멤버 함수가 추가된다. 이 멤버 함수는 Clo 클래스의 추상 멤버 함수

code()를 대신할(override) 수 있도록 code() 함수라 이름 붙인다.

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

위의 클래스 정의에서 멤버 변수의 타입(혹은 클래스)으로 Object를 지정했다. Object 클래스는 Java 언어에서 정의된 모든 클래스의 상위 클래스로 정의되어 있다. 따라서 객체가 어떤 클래스에 속하는지 상관없이 임의의 멤버 변수에 이 객체를 저장할 수 있다. 함수형 언어의 다형성(polymorphism)으로 인해 하나의 멤버 변수 f_i에 여러 다른 타입의 객체가 저장될 수 있다. Object 타입의 멤버 변수에 저장된 객체를 다루기 전에 Object 타입을 적절한 타입으로 변환하는(cast) 과정이 필요하다. 다음 장에서 제시할 L-code 컴파일러에서 보여주듯이 타입 변환 과정은 컴파일 시점에 모두 결정 가능하다.

3.2 실행시간 시스템 (runtime system)

실행시간 시스템으로 클래스 G를 두고 필요한 요소들을 이 클래스의 정적(static) 멤버 변수 node, tag, loopflag, sp, bp, stk로 정의한다.

```
public class G {
    public static Object node;
    public static int tag;
    public static boolean loopflag;

    public static int sp, bp;
    public static Object[] stk;
        ...
}
```

L-code 명령어 JMPK와 GETNT에서 힙 클로져의 주소와 태그를 전달하고 전달받는다. 아래의 GETNT에 대한 환원 규칙에서 레지스터 x와 t를 통해 주소와 태그를 전달받는다.

$$(\mathbf{GETNT}\ (\lambda x.\lambda t.C))\ x\ t\ s\ h \Rightarrow C\ \mu_0[x \mapsto x, t \mapsto t]\ s\ h$$

GETNT를 컴파일하여 얻은 Java 코드에서 클래스 G의 멤버변수 G.node와 G.tag를 통해 주소와 태그를 받는다. 이 주소와 태그는 JMPK를 컴파일하여 얻은 Java 코드에서 저장한 것이다.

L-machine의 레지스터 파일 μ는 Java 언어의 변수 선언과 가시 영역 규칙(scope rule)으로 구현할 수 있다. 별도의 특별한 표현을 필요로 하지 않는다. 스택은 Object 타입의 배열 G.stk로 표현하고 스택의 맨 위 원소를 G.sp로 가리키고 스택의 맨 하위 원소를 G.bp로 가리킨다. L-machine의 힙은 JVM이 제공하는 힙으로 표현한다.

스택을 배열로 표현할 때 두 가지 고려해야 할 점이 있다. 첫째, 스택에서 객체를 가져올 때 그 객체가 있던 배열 원소는 null 값으로 채워야한다. JVM의 가비지수집기는 스택을 표현하는 배열을 특별하게 취급하지 않으므로 가비지수집이 필요할 때마다 배열의 모든 원소를 탐색한다. 만일 null 값으로 채운 다음에 그 자리에 있던 객체가 다른 곳에서 더 이상 사용되지 않으면 그 객체가 차지하던 메모리를 메모리관리 시스템에서 회수할 수 있다. 둘째, Object 타입의 배열은 Java 언어에서 참조 데이타 타입(reference data type)[5]이라 부르는 값만을 저장할 수 있다. 클래스로 정의된 모든 타입은 참조 데이타 타입이다. 따라서 정수형, 실수형, 문자형과 같은 기본형 자료 타입의 값은 이 스택을 통해서 전달할 수 없다. 정해진 클래스의 정적 멤버 변수를 통해 전달하도록 구성하였다.

3.3 갱신 (updates)

보통의 환경에서 갱신을 구현하는 방법은 주어진 주소의 메모리 영역에 갱신할 값을 직접 쓰는 것이다. 이후 그 주소를 참조하면 갱신 전에 그 주소에 있는 클로져 대신 갱신한 값을 읽게 된다. JVM 환경에서는 특정 주소에 놓은 객체 위에 또 다른 객체를 쓰는 것은 가능하지 않다. 따라서 갱신을 위해서 다음의 클래스를 도입한다.

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

갱신이 필요한 클로져에 대한 객체를 만들 때마다 Ind 클래스의 객체를 만들어 이 객체의 ind 멤버 변수가 그 클로져 객체를 가리키게 한다. 나중에 이 클로져 객체를 갱신해야할 때 이 ind 멤버 변수가 갱신하고자 하는 클로져 객체를 가리키게 한다. 갱신에 대한 이러한 아이디어는 다음 장에서 제시할 L-code 컴파일러에서 PUTC와 UPDC 명령어에 대한 컴파일 규칙에서 보여준다.

갱신 후 Ind 클래스의 객체는 여전히 남아 있어 메모리를 불필요하게 점유하는 문제가 발생할 수 있다. 보통의 구현 환경에서는 이 객체를 특별하게 간주하여 가비지수집기에서 제거하는 방법으로 해결한다. Ind 클래스는 JVM의 입장에서는 특별한 클래스가 아니므로 동일한 방법으로 이 문제를 해결할 수 없다. 본 논문에서의 STGM 구현 방법에서는 두 개 이상의 Ind 클래스 객체가 연달아 이어지는 형태를 갖지는 않지만 한 개의 Ind 클래스 객체만으로도 불필요하게 메모리를 점유하는 문

제를 일으킬 수 있다. 이 문제는 지연 함수형 언어의 갱신 연산과 관련된 것으로 지연 함수형 언어를 JVM으로 컴파일하는 모든 방법에 공통적인 것이다.

3.4 테일 호출(tail call)

STGM에서 제어 흐름은 대부분 테일 호출을 통해 이뤄진다. 보통의 구현 환경에서 테일 호출은 단순히 분기 명령어로 쉽게 구현할 수 있다. 하지만 JVM 환경에서는 테일 호출에 관련된 명령어를 지원하지 않으므로 테일 호출을 구현하는 별도의 방법이 필요하다. 클래스 G에 다음의 정적 멤버 함수로 구현한 간단한 인터프리터를 둔다.

public static void *loop* (*Clo c*) { while(*loopflag*) $c = c.ind.code()$; }

앞서 제시한 STG 컴파일러에서 만들어내는 L-code 명령어들은 JMPK, JMPC, CASE 중 하나로 끝난다. L-code 컴파일러에서 이 명령어들을 다음에 실행할 객체를 리턴(return)하는 Java 문장으로 변환한다. 리턴 후 위 인터프리터의 while 문 내의 할당문(assignment statement)이 실행되고 while 문에서 반복 조건을 검사한 다음 리턴한 객체의 code() 함수를 호출하게 된다. 인터프리터에서 loopflag는 실행시간 시스템을 나타내는 클래스 G의 정적 멤버 변수로 진행 여부를 지정한다.

4. 컴파일

L-code 컴파일러는 단 하나의 let 블록에 모든 바인딩이 있는 L-code 프로그램을 입력받아 [그림 4]에서 제시한 L-code 명령어 컴파일 규칙에 따라 Java 클래스를 낸다. 본 논문에서 고려하는 기본 원칙은 각 클로져 타입에 대해 하나의 클래스를 내는 것이다. 이 클래

$$
\begin{array}{ll}
J[\![\text{JMPC x}]\!] & = G.node = x;\ \text{return}\ (Clo)x \\
J[\![\text{JMPK x y t}]\!] & = G.node = y;\ G.tag = t;\ \text{return}\ (Clo)x \\
J[\![\text{GETN}\ (\lambda x.C)]\!] & = Object\ x = G.node;\ J[\![C]\!] \\
J[\![\text{GETNT}\ (\lambda x.\lambda t.C)]\!] & = Object\ x = G.node;\ \text{int}\ t = G.tag;\ J[\![C]\!] \\
J[\![\text{GETA}\ (\lambda\langle l, x\,\overline{w}\rangle.C)]\!] & = J[\![C]\!] \\
J[\![\text{PUTARG}\ \overline{x}_n\ C]\!] & = push\ x_n; \ldots\ push\ x_1;\ J[\![C]\!] \\
J[\![\text{GETARG}\ (\lambda\overline{x}_n.C)]\!] & = Object\ x_1,\ldots,x_n;\ pop\ x_1; \ldots\ pop\ x_n;\ J[\![C]\!] \\
J[\![\text{PUTK}\ \langle l, \overline{x}_n\rangle\ C)]\!] & = C_l\ o = \text{new}\ C_l();\ o.f_1 = x_1; \ldots\ o.f_n = x_n;\ push\ o;\ J[\![C]\!] \\
 & \quad \text{where } o \text{ fresh} \\
J[\![\text{GETK}\ \lambda\langle l, \overline{x}_n\rangle.C)]\!] & = Object\ o;\ pop\ o; \\
 & \quad Object\ x_1 = ((C_l)o).f_1, \ldots, x_n = ((C_l)o).f_n;\ J[\![C]\!] \\
 & \quad \text{where } o \text{ fresh} \\
J[\![\text{REFK}\ (\lambda x.C)]\!] & = Object\ x = G.stk[G.sp];\ J[\![C]\!] \\
J[\![\text{SAVE}\ (\lambda s.C)]\!] & = \text{int}\ s = G.bp;\ J[\![C]\!] \\
J[\![\text{DUMP C}]\!] & = G.bp = G.sp;\ J[\![C]\!] \\
J[\![\text{RESTORE s C}]\!] & = G.bp = s;\ J[\![C]\!] \\
J[\![\text{PUTC}\ \overline{x_i \overset{\pi_i}{=} \langle l_i, \overline{w^i_{j\,m_i}}\rangle_n}\ C]\!] & = \ldots\ alloc_i;\ \ldots\ assign^i_j; \ldots\ J[\![C]\!] \\
 & \quad \text{where } alloc_i \equiv C_{l_i}\ x_i = \text{new}\ C_{l_i}() \qquad \text{if } \pi_i = n \\
 & \qquad\qquad \equiv Ind\ x_i = \text{new}\ Ind(); \qquad \text{if } \pi_i = u \\
 & \qquad\qquad\quad x_i.ind = \text{new}\ C_{l_i}() \\
 & \qquad assign^i_j \equiv x_i.f_j = w^i_j \qquad \text{if } \pi_i = n \\
 & \qquad\qquad \equiv ((C_{l_i})x_i.ind).f_j = w^i_j \qquad \text{if } \pi_i = u \\
J[\![\text{GETC y}\ (\lambda\langle l, \overline{x}_n\rangle.C)]\!] & = Object\ x_1 = ((C_l)y).f_1; \ldots; x_n = ((C_l)y).f_n;\ J[\![C]\!] \\
J[\![\text{UPDC x}\ \langle l, \overline{w}_n\rangle\ C]\!] & = C_l\ o = \text{new}\ C_l();\ o.f_1 = w_1;\ \ldots\ o.f_n = w_n; \\
 & \quad ((Clo)x).ind = o;\ J[\![C]\!] \quad \text{where } o \text{ fresh} \\
J[\![\text{PUTP}\ \overline{\langle l_i, y\rangle_n}\rangle\ (\lambda x.C)]\!] & = \text{switch}(G.sp - G.bp)\ \{ \\
 & \qquad \text{case } 0 : C_{l_1}\ x = \text{new}\ C_{l_1}();\ x.f_1 = y;\ \text{break}; \ldots \\
 & \qquad \text{case } n-1 : C_{l_n}\ x = \text{new}\ C_{l_n}();\ x.f_1 = y; \\
 & \qquad\qquad pop\ x.f_2; \ldots\ pop\ x.f_n;\ \text{break}; \\
 & \quad \}\ J[\![C]\!] \\
J[\![\text{CASE t}\ \overline{c_i \to \langle l_i, x_0\,\overline{x}_i\rangle}\ \text{default} \to \langle l_d, x_0\,\overline{x}_d\rangle]\!] & \\
 & = \text{switch}(t)\ \{ \ldots\ \text{case}\ c_i : J[\![C_{l_i}]\!]\ \ldots\ \text{default} : J[\![C_{l_d}]\!]\} \\
J[\![\text{ARGCK}\ n\ C_1\ C_2)]\!] & = \text{if}(n > G.sp - G.bp)\ \text{then}\ \{\ J[\![C_1]\!]\ \}\ \text{else}\ \{\ J[\![C_2]\!]\ \} \\
J[\![\text{STOP y}]\!] & = G.loopflag = \text{false};\ \text{return null}; \\
 & \\
push\ X & = G.sp{+}{+};\ G.stk[G.sp] = X \\
pop\ X & = G.sp{-}{-};\ X = G.stk[G.sp+1];\ G.stk[G.sp+1] = \text{null}
\end{array}
$$

그림 4 L-code 컴파일러의 컴파일 규칙

스는 클로져의 자유 변수를 표현하는 멤버 변수와 클로져의 코드에 해당하는 멤머 함수를 갖는다.

L-code 컴파일러를 위해 각각의 바인딩에 대해 하나의 클래스를 내는 간단한 드라이버(driver)가 필요하다. 편의상 클로져의 자유 변수는 주석의 형태로 주어진다고 가정한다. 드라이버는 바인딩 $l = (\overline{w}).C$를 받아 이름 C_l의 클래스를 만들고, 이 클래스에 $\overline{w}$를 멤버 변수로 두고, 멤버 함수 code()의 몸체에는 [그림 4]에서 제시한 컴파일 규칙을 C에 적용하여 만든 Java 문장으로 채운다. 궁극적으로 Java 클래스로 이뤄진 Java 프로그램을 얻을 수 있다. 이 클래스는 앞서 설명한 몇몇 클래스와 함께 Java 컴파일러로 컴파일하여 JVM에서 실행할 수 있는 Java 바이트 코드 프로그램을 만들 수 있다.

컴파일 규칙에 의하면 L-code 프로그램의 레지스터는 Java 프로그램의 지역 변수(local variable)로 표현해 변수 이름으로 레지스터를 다룬다. 이는 레지스터 집합에 대해 별도의 배열을 두어 배열 이름과 적절한 오프세트(offset)로 레지스터를 다루는 구현 방법보다 효율적이다.

컴파일 규칙 J에 의하면 L-code 프로그램에서 $\ldots\ (\lambda\overline{x}.\ \ldots)$ 형태인 경우 일반적으로 $\overline{x}$에 대해 Java 프로그램에서 동일한 이름의 Object 타입의 지역 변수를 선언한다. GETA에 대한 J 규칙에서와 같이 예외적으로 지역변수를 선언하지 않아도 되는 경우가 있다. 이유는 필요한 지역 변수가 이전에 이미 선언되어있기 때문이다. STG 컴파일러에 의하면 CASE 명령어가 위치한 클로져와 GETA가 위치한 클로져는 다르지만, L-code 컴파일러에 의하면 두 클로져를 합해서 하나의 클래스를 만든다. 이때 GETA가 위치한 클로져의 자유 변수는 CASE가 위치한 클로져의 자유 변수 중의 하나이다. 또한 CASE에 대한 J 규칙에 의하면 GETA가 위치한 L-code를 합하여 하나의 Java 클래스를 만들기 때문에 GETA에서 선언하고자 했던 지역 변수는 CASE 문 이전의 L-code 명령어를 컴파일 할 때 이미 선언되어있으므로 불필요한 중복 선언을 하지 않는다.

SAVE와 RESTORE에 대해서는 이 명령어에 대한 환원 규칙이 정의한 것처럼 전체 스택을 저장하고 복구하는 것은 아니다. 스택의 최하위 원소를 가리키는 포인터 G.bp를 저장하고 복구하는 것으로 이 환원 규칙에서 정의한 것을 효율적으로 표현할 수 있다. 이 기법은 Peyton Jones [1]에 의해 제안되었다. SAVE(λs.C)를 컴파일할 때 지역 변수 s의 타입은 Object 타입대신 int로 정한다. 왜냐하면 int 타입의 G.bp의 값이 s에 저장될 것이기 때문이다.

다음은 2.4절에서 제시한 l_a의 코드를 J 규칙에 의해 컴파일한 결과이다. 이 컴파일 예제는 L-code의 각 바인딩이 대해 하나의 Java 클래스로 변환됨을 보여준다.

```
public class C_{l_a} extends Clo {          // l_a =
    public Object f1, f2;                   // {g, x}.
    public Clo code() {
        Object z = G.node;                  // GETN λz.
        Object g = ((C_{l_a})z).f1;         // GETC λ⟨l_a, g x⟩.
        Object x = ((C_{l_a})z).f2;
        int s = G.bp;                       // SAVE λs.
        C_{l_upd} o = new C_{l_upd}();      // PUTK ⟨l_upd, z s⟩
        o.f1 = z; o.f2 = s;
        G.sp ++; G.stk[G.sp] = o;
        G.bp = G.sp;                        // DUMP
        G.sp ++; G.stk[G.sp] = x;           // PUTARG x
        return (Clo)g;                      // JMPC g
    }
}
```

5. 벤치마킹

본 논문에서 Meehan과 Joy[6] 가 사용한 5개의 작은 벤치마크 프로그램(fib 30, edigits 250, prime 500, soda, queen 8)에 제안한 컴파일러를 적용해 실험했다. 실험 환경은 296 MHz 프로세서, 768 Mbytes 메모리, Solaris 2.5.1에 기반한 SUN UltraSPARC-II 워크스테이션이다. Haskell 전단부로 GHC 4.04를 사용하였고, Java 프로그램 컴파일러로 Sun JIT 1.2.2를 사용하였다. 성능 비교를 위해 GHC4 4.04, Hugs98 Feb-2000을 사용하였다. 최적화된 STG 프로그램을 얻기 위해 GHC 4.04 컴파일러에 -O2 옵션을 주었다.

표 1 코드 크기 (바이트)

프로그램	GHC	JIT	JIT(최적화)
fib	268,028	24,830 (037 classes)	18,610 (034 classes)
edigits	283,092	113,913 (135 classes)	64,327 (092 classes)
prime	280,248	74,161 (096 classes)	50,025 (070 classes)
queen	274,816	108,705 (134 classes)	75,619 (101 classes)
soda	302,972	335,873 (388 classes)	185,496 (203 classes)

표 2 실행 시간 (초)

프로그램	GHC	Hugs	JIT	JIT(최적화)
fib	0.18s	106.48s	25.70s	5.72s
edigits	0.16s	3.10s	9.15s	2.42s
prime	0.14s	3.30s	86.38s	1.97s
queen	0.07s	5.79s	5.35s	2.29s
soda	0.03s	0.41s	2.26s	1.59s

5.1 STG 프로그램 최적화

STGM을 사용해서 얻는 장점 중 하나는 기존에 개발된 STG 프로그램의 최적화 방법을 적용할 수 있는 점이다. 먼저 GHC 컴파일러에 -ddump-stg 옵션을 주어

Haskell 프로그램으로부터 STG 프로그램을 얻어낸다. 본 논문에서 구현한 컴파일러는 이 STG 프로그램을 받아 제안한 컴파일 규칙에 따라 Java 클래스로 변환한다. 이상적으로 본 논문에서 제안한 컴파일러는 기존의 Haskell 컴파일러의 후단부로 볼 수 있다.

5.2 결 과

[표 1]은 각 경우의 코드 크기를 보여준다. GHC의 경우 동적 연결용으로 컴파일하고 불필요한 심볼 테이블은 제거한 코드로부터 크기를 측정한 것이다. Hugs의 경우 코드는 Hugs 환경하에서 메모리 상에 생성이 되므로 코드 크기를 측정할 수 없다. 본 논문에서 제안한 컴파일러의 경우 각 Haskell 프로그램으로부터 나온 클래스 파일과 실행시간 시스템을 위한 클래스 파일을 합해 크기를 측정한 것이다. 이 경우 클래스 파일 크기와 함께 클래스의 개수를 괄호 안에 기재했다. 실행시간 시스템은 4개의 클래스, 2972 바이트로 구성되어 있다.

[표 2]는 실행 시간을 보여준다. 실행 시간은 사용자 시간과 시스템 시간을 합한 것이고 유닉스 시간 측정 명령어 time으로 측정한 것이다. 실행 시간은 5회 실행 중 가장 짧은 시간을 선택했다. Hugs의 경우 실행 시간은 Hugs 환경에서 제공하는 방법으로 측정한 것으로 파싱, 타입 체킹, 코드 생성 시간은 포함되지 않는다.

5.3 토 의

[표 1]에 의하면 각각의 경우 비교적 많은 개수의 Java 클래스로 구성되어 있다. 이는 각 클로져 마다 하나의 클래스를 만드는 전략에서 비롯된 것이다. 특히 최적화 변환을 적용하지 않은 soda 프로그램의 경우 그 크기가 GHC의 바이너리 실행 파일의 크기보다 크다. 이는 GHC에서 최적화하지 않고 컴파일 할 때 프로그램 내에서 리스트로 표현된 이차원 배열로부터 지연 식(thunk)이 많이 만들어지는 것에서 비롯된다. 이 지연식이 하나의 클래스로 변환되므로 클래스 개수와 전체 클래스 크기가 증가한 것이다. Wakeling[7]에서 여러 공통된 클래스를 하나의 클래스로 합하여 전체 클래스 개수와 크기를 줄였던 것과 같이 제안한 컴파일러에서 내는 클래스 파일 개수를 유사하게 줄일 수 있다. 즉 각각의 클래스 중 동일한 타입과 개수의 멤버 변수를 갖는 클래스는 하나의 클래스로 합할 수 있다. L-code 컴파일러를 변경하여 5개의 벤치마크 프로그램에 대해서 전체적으로 그 크기를 반 정도로 줄일 수 있었고 클래스 파일 개수는 20개 안팎으로 줄일 수 있었다.

[표 2]에 의하면 생성된 Java 프로그램의 실행시간은 GHC 컴파일러에 의해 생성된 바이너리의 실행시간과 Hugs에 의한 실행시간의 사이에 위치했다. 우선 GHC 컴파일러에 의해 생성된 바이너리 보다 실행시간이 느린 점은 예측했던 바이다. 왜냐하면 JVM은 메모리 할당에 시간이 많이 걸리고, Haskell 프로그램에서 검증된 타입 안정성으로 인해 불필요한 실행시간 검사 (null 포인터 검사)를 하고, 빈번한 테일 호출에 대해 구현 방법이 훨씬 비효율적이고, 스택을 배열로 표현하여 배열 원소를 접근할 때마다 배열의 경계 검사를 하고, 스택에서 원소를 가져간 후 항상 null을 저장해야 하기 때문이다.

GHC의 최적화 변환을 적용하면 생성된 Java 프로그램의 실행시간은 Hugs의 실행시간보다 짧았다. soda 프로그램의 경우 1.59초로 Hugs에서의 실행 시간인 0.41초보다 길다. 이는 JVM이 실행되어 Java 프로그램이 비로소 실행되기까지 대략 0.68~0.74초가 걸리므로 Hugs의 경우보다 더 빨라질 수 없었다. STG 컴파일러에서 case문에 기본 연산자가 위치한 경우 컴파일러 규칙을 개량하여 전체 벤치마크 프로그램에 대해 실행시간을 평균적으로 현재 실행시간의 반 정도까지 줄일 수 있었다.

GHC에서 제공한 STG 프로그램 최적화 방법은 모든 프로그램의 실행 시간에 영향을 미쳤다. 특히 prime 프로그램의 경우 86.38초에서 1.97초로 매우 효과적이었다. 지연계산 기반 함수형 언어의 경우 힙 할당량이 수행시간에 영향을 미치므로 전체 힙 할당량을 비교하여 수행시간의 변화를 설명할 수 있다. 최적화 되지 않은 경우 전체 힙 할당량은 2,603,827,792 바이트로 최적화 된 경우 35,061,856 바이트보다 74.3배 더 많은 힙을 할당했다.

6. 관련 연구

STGM에 기반한 연구로 Tullsen[3]와 Vernet[4]이 있다. 두 사람 모두 자신의 컴파일러에 대한 구체적인 명세 없이 서술적이고 예를 중심으로 설명하였다. 이런 설명만으로 그들의 컴파일러를 동일하게 다시 구현해 볼 수 없다. 또한 그들이 제안한 컴파일러로 생성한 Java 프로그램의 성능에 대한 언급이 없고 STG 프로그램에 대한 최적화 변환의 영향에 대한 언급 역시 없다.

$<\nu,G>$-machine과 G-machine을 기반으로 하는 연구가 있었다. Wakeling[7]이 제안한 컴파일러는 $<\nu,G>$-machine을 기반으로 한다. 이 컴파일러는 스택을 여러 작은 블록의 연결 리스트로 표현한다. 이는 $<\nu,G>$-machine의 함수 호출 방법인 eval-apply 모델에서 비롯된다. 그리고, 클래스의 효율적인 표현으로 비교적 큰 벤치마크 프로그램에 대해서도 생성된 클래스 개수와 크기를 크게 줄일 수 있음을 보였다. 특히 클래스 개수와 크기가 전체 실행시간에 영향을 준다고 주장하였다.

그의 컴파일러에서 생성한 Java 프로그램의 실행시간은 Hugs에서 실행시간과 비슷함을 보였다. 본 논문에서 제안한 컴파일러와 그의 컴파일러가 생성하는 Java 프로그램의 성능을 비교하지는 않았지만 본 논문에서 선택한 STGM 기반 방법이 더 적합하다고 생각된다. 왜냐하면 그의 컴파일러는 STG 프로그램 최적화 변환을 이용할 수 없기 때문이다. 두 연구의 성능 비교에 관한 연구가 필요하다.

Meehan과 Joy[6]는 하나의 클래스만을 두어 각 클로져의 코드를 클래스의 멤버 함수로 표현하고 Java 라이브러리에서 제공하는 리플렉션 패키지(reflection package)를 사용하여 멤버 함수를 다룬다. 그들의 컴파일러에서 생성한 Java 프로그램의 실행시간은 Hugs에서보다 느리다는 실험 결과를 제시했다.

최근에 Haskell 메일링 리스트에서 Meijer, Perry, Gill이 GHC에 대한 Java 후단부에 관한 작업을 진행중이라고 언급한 바 있다. 하지만 아직 그들의 작업에 대한 어떠한 문서도 찾을 수 없었다[8].

스트릭트 함수형 언어(strict functional language)에 대해서도 유사한 연구가 진행되었다. Benton, Kennedy, Russel[9,10]은 Standard ML(SML)프로그램으로부터 Java 프로그램을 생성하는 컴파일러를 개발한 바 있다. 방대한 최적화를 통해 Java 프로그램의 성능을 향상시키고자 하는 점은 본 논문에서 채택한 방법과 유사하다. 또한 그들은 SML과 Java를 혼합 프로그래밍 할 수 있는 방법에 관해서도 연구하였다.

7. 결론 및 향후 연구

본 논문에서는 중간 언어 L-code를 정의하고 STG 컴파일러와 L-machine을 제시하여 STGM에 대한 간결하고 완전한 명세를 제시하였다. 이를 통해 정확하게 STGM을 구현할 수 있었다. L-code의 각 명령어는 몇몇 Java 문장으로 쉽게 변환되고, L-code 클로져는 Java의 클래스로 쉽게 변환된다. Java 언어가 객체 지향 언어이지만 이 패러다임을 위한 별도의 특징을 L-code에 부여하지 않고도 STG 프로그램을 Java 프로그램으로 간결하게 변환하는데 도움을 주었다. 제시한 명세와 표현 방법에 기초하여 L-code 컴파일러를 제시하였다.

본 논문에서 택한 추상기계는 STGM이므로 이미 개발된 STG 프로그램에 대한 최적화 변환을 그대로 이용하여 Java 프로그램의 성능을 향상시킬 수 있다. 5개의 작은 벤치마크에 대해 실험한 결과 제안한 컴파일러는 이들 최적화 변환을 적용했을 때 빠른 성능의 Java 프로그램을 생성함을 확인하였다.

향후 연구로 다음과 같은 사항이 있다. 먼저 현재 채택한 Java 표현 방법이 비교적 단순하다. 따라서 더욱 효과적인 표현 방법을 고안하여 생성된 Java 프로그램의 성능을 더욱 높이는 노력이 필요하다. 둘째로 더 큰 벤치마크 프로그램에 제안한 컴파일러를 적용할 수 있도록 현재의 프로토타입 구현을 더욱 발전 시켜야 한다.

참 고 문 헌

[1] S. L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine. Journal of Functional Programming, 2(2):127-202, April 1992.

[2] S. L. Peyton Jones and A. L. M. Santos. A Transformation-based Optimiser for Haskell. Science of Computer Programming, 32(1-3):3-47, 1998.

[3] M. Tullsen. Compiling Haskell to Java. 690 Project, Yale University, September 1997.

[4] A. Vernet. The Jaskell Project. A Diploma Project, Swiss Federal Institute of Technology, February 1998.

[5] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. The Java (tm) Language Specification, Second Edition. Addison Wesley, June 2000.

[6] G. Meehan and M. Joy. Compiling Lazy Functional Programs to Java Bytecode. Software-Practice and Experience, 29(7):617-645, June 1999.

[7] D. Wakeling. Compiling Lazy Functional Programs for the Java Virtual Machine. Journal of Functional Programming, 9(6):579-603, November 1999.

[8] S. L. Peyton Jones. A Java Back End for Glasgow Haskell Compiler. The Haskell Mailing List haskell@haskell.org (http://www.haskell.org/mailinglist.html), May 2000.

[9] N. Benton and A. Kennedy. Interlanguage Working Without Tears: Blending SML with Java. In Proceedings of the 4th ACM SIGPLAN Conference on Functional Programming, pages 126-137, 1999.

[10] N. Benton and A. Kennedy, and G. Russel. Compiling Standard ML to Java Byte-codes. In Proceedings of the 3rd ACM SIGPLAN Conference on Functional Programming, pages 129-140, 1998.

최 광 훈

1994년 한국과학기술원 전산학과 졸업. 1996년 한국과학기술원 전산학과 석사학위 취득. 1996년 ~ 현재 한국과학기술원 전산학과 박사과정. 관심분야는 프로그래밍언어, 컴파일러, 함수형 언어 등

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