자바가상기계를 대상으로 하는 지연계산기반 함수형 언어 컴파일러의 설계 및 구현

최광훈 임현일 한태숙
한국과학기술원 전자전산학과
{khchoi, hilim}@pllab.kaist.ac.kr, han@cs.kaist.ac.kr

Design and Implementation of a Lazy Functional Language Compiler for Java Virtual Machine

Kwanghoon Choi Hyun-il Lim Taisook Han
Dept. of EE & CS, KAIST

요 약

본 논문에서는 지연계산기반 함수형 언어 프로그램을 받아 Java 프로그램을 목적 코드로 생성하는 컴파일러를 설계하고 구현한다. 이 컴파일러는 제한된 형태의 함수형 언어 Shared Term Graph(STG)를 입력언어로 하는 추상기계 Spineless Tagless G-Machine(STGM)을 수행 모델로 한다. 본 논문에서는 명령형 언어 L-code를 제안하고 이를 입력언어로 하는 새로운 형태의 STGM인 L-Machine을 제안한다. STG 언어를 L-code로 변환하는 컴파일러와 L-code를 Java로 변환하는 컴파일러를 설계하고 구현함으로써 원하는 컴파일러를 구성한다. 그리고 Glasgow Haskell 컴파일러를 전단부로 하여 지연계산기반 함수형 언어 Haskell로 작성된 프로그램들을 컴파일하여 STG 프로그램으로 변환한 다음 본 논문에서 구현한 컴파일러로 이를 Java 프로그램으로 변환한다. 변환된 Java 프로그램을 Sun JIT 컴파일러로 컴파일하여 수행한 성능 평가 결과를 제시한다.

1. 서론

함수형 언어를 자바가상기계[7]를 목적대상으로 컴파일 하는 동기는 컴파일 된 코드를 자바가상기계가 설치된 임의의 기계에서 실행할 수 있는 이동성(mobility)과 상대적으로 많은 라이브러리를 포함하는 자바 언어와의 혼합 프로그래밍(interlanguage)의 가능성에서 비롯된다.

본 논문에서는 Spineless Tagless G-Machine(STGM)을 수행 모델로 하여 지연계산기반 함수형 언어를 자바 언어로 변환하는 컴파일러를 설계하고 구현하는 것이다. STGM은 지연계산기반 추상기계 중 가장 발전된 형태이고[4], 이 추상기계를 사용하면 입력 언어 Shared Term Graph(STG)에 대한 기존의 많은 최적화 변환 방법을 적용할 수 있는 장점이 있다[6].

본 논문에서 제안하는 컴파일러는 함수형 언어 형태의 STG를 입력언어로 하는 기존의 STGM을 명령형 언어 형태 L-code를 입력언어로 하는 새로운 형태의 STGM인 L-machine을 기반으로 설계한다. 즉, 이 컴파일러는 STG 프로그램을 L-code 프로그램으로 변환하고, L-code 프로그램을 자바 프로그램으로 변환하는 2단계로 구성된다. 이와 같이 STG를 임의의 목적 대상 명령형 언어로 변환하여 STGM을 구현하는 작업과 특정 목적 언어로 선택한 자바 언어로 이를 표현하는 작업을 분리하여 전체 컴파일러를 구체적으로 제시한다.

2. 관련연구

STGM에 기반하여 지연계산기반 함수형 언어를 자바 언어로 컴파일하는 방식은 Tullsen[9]과 Vernet[2]이 시도한 바 있다. 이들은 STG 프로그램을 직접 자바 프로그램으로 변환하려 했는데 이 경우 STGM을 구현하는 작업과 자바 언어로 이를 표현하는 작업이 하나로 구성된다. 두 연구 모두 변환 작업에 대한 구체적인 규칙을 제시하지 않았고, 각자의 변환 결과로 얻은 자바 프로그램의 성능 평가에 대한 언급이 없다. 또한, STG에 적용가능한 프로그램 최적화 변환의 효과에 관한 언급이 없다.

3. Spineless Tagless G-Machine

STGM은 입력 언어인 STG 언어, STG를 L-code로 변환하는 컴파일러, L-code, L-machine으로 설명할 수 있다. 이 절에서는 L-code와 L-machine만으로 STGM을 설명하고 나머지 사항은 저자의 확장된 논문[1]에 자세히 제시되어 있다.

아래의 예는 Haskell로 쓰여진 S 컴비네이터와 이를 STG 언어로 변환한 것이다.

(Haskell) : let $s\ f\ g\ x = f\ x\ (g\ x)$ in ...

(STG) : let $s = {}^{n}\lambda\ f\ g\ x.\ let\ h = {}^{u}g\ x$ in $f\ x\ h$
in ...

다음은 L-code의 구문이다. L-code는 구문 상에서의 바인딩 관계를 람다텀에서의 바인딩 관계로 표현하는 고차원 어셈블리 언어(higher-order assembly language[5]) 스타일로 설계되었다. 본 논문에서는 기호 $\overline{obj}_n = obj_1 \ldots obj_n$ 를 사용한다.

위의 STG 언어로 표현된 S 컴비네이터를 L-code로 변환하면 다음과 같다. 구체적인 변환 규칙은 [1]을 참조한다.

* 본 연구는 첨단정보기술 연구센터를 통하여 과학재단의 지원을 받았음.

```
x,y,z,w,t,s                                                    register
l                                                              label
C ::= let l̄ = C̄ in C                                           L-code
    | JMPC x          | JMPK x y t          | JMPM t t→⟨l,{x,w̄}⟩
    | GETV (λx.C)     | GETVT (λx.λt.C)     | GETVA (λ⟨l,{x,w̄}⟩.C)
    | PUTA x̄ C         | GETA (λx̄.C)         | ARGCK n C_mp C
    | PUTK ⟨l,{x̄}⟩ C   | GETK_l (λx̄.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,{x}⟩ (λy.C) | GETP x (λ⟨l,{w0,w̄}⟩.C) | STOP x
    | GETCA x (λ⟨l,{w̄}⟩.C) | GETCV x (λy.C)
```

```
(L−code) :
let l_upd = ... in
let l_s = GETV (λz.GETC z (λ⟨l_s,{}⟩.ARGCK 3 (...)(\GETA (λf g x.
        let l_h = GETV (λz.GETC z (λ⟨l_h,{g,x}⟩.SAVE (λs.
                  PUTK ⟨l_upd,{z,s}⟩(DUMP (PUTA x (JMPC g))))))
        in PUTC h = "⟨l_h,{g,x}⟩ (PUTA x h (JMPC f))))))
in ...
```

각 L-code 명령어의 의미는 L-machine에 의해 정의된다[1]. L-machine은 상태전이 규칙으로 정의되고, 임의의 시점에서 상태는 일반적으로 L-code 명령어 C, 레지스터 파일 μ, 스택 s, 힙 h로 구성된다. 레지스터 파일은 레지스터 이름을 힙 주소, 라벨, 태그, 스택과 바인딩하는 실행시간 자료구조이다. L-code에서 클로져(closure) $\langle l, \rho \rangle$는 라벨과 환경의 쌍으로 나타내는데 여기서 환경은 힙 주소의 집합 $\rho = \{\bar{x}\}$이다.

STG-to-L-code 변환 규칙[1]에 의하면 변환 결과로 얻은 L-code의 모든 let에 바인딩된 C는 닫혀있으므로(외부에 선언된 레지스터를 사용하지 않으므로) 모두 맨 상위 let 블록으로 모을 수 있다(hoisting). 앞의 L-code 예제는 호이스팅 변환을 통해 다음과 같은 코드로 변환 가능하다.

```
(L−code) :
let l_upd = ...
    l_s = GETV (λz.GETC z (λ⟨l_s,{}⟩.ARGCK 3 (...)(\GETA (λf g x.
          PUTC h = "⟨l_h,{g,x}⟩ (PUTA x h (JMPC f))))))
    l_h = GETV (λz.GETC z (λ⟨l_h,{g,x}⟩.SAVE (λs.
          PUTK ⟨l_upd,{z,s}⟩(DUMP (PUTA x (JMPC g))))))
in ...
```

이 예에서 볼 수 있듯이 호이스팅 변환 결과로 얻은 L-code는 하나의 let식이고 let에 바인딩된 어떤 L-code C도 내부에 let식을 포함하지 않는다. let에 바인딩된 C는 L-code에서 클로져에 상응한다. 위의 l_s에 대한 바인딩이 있으므로 $\langle l_s, \{\} \rangle$ 형태의 클로져를 구성할 수 있고, l_h에 대한 바인딩이 있으므로 $\langle l_h, \{y_g, y_x\} \rangle$ 형태의 클로져를 구성할 수 있다.

4. STGM 구성요소를 자바 언어로 표현

본 절에서는 STGM 구성요소를 자바 언어로 표현하는 방법에 대해서 설명하고, 다음 절에서 이를 기반으로 L-code를 자바 언어로 변환하는 방법에 관해서 설명한다.

모든 클로져는 아래 자바 클래스의 하위 클래스로 표현한다. 자바 언어에서는 모든 클래스의 상위 클래스로 Object 클래스가 정의되어 있으므로 논문에서 사용하는 클래스 계층은 Object, Clo, 각 클로져에 대한 클래스이다. 필드 self는 갱신(update)을 위해 추가되었고, 메소드 code()는 클로져의 L-code를 변환해 얻은 자바 코드로 정의된 메소드로 오버라이딩(overriding)된다.

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

지연계산기반 함수형 언어는 지연된 식을 계산한 후 결과 값을 식에 대치하여 나중에 이 식의 값이 필요한 경우 중복 계산하지 않고 결과 값을 이용하는 특징이 있다. 이때 결과 값을 식에 대치하는 과정을 갱신(update)이라 한다. 기존의 목적 기계와 달리 자바가상기계에서는 임의의 객체 위에 다른 객체를 덮어 쓸 수 있는 방법이 없으므로 앞서 정의했던 self라는 필드와 다음의 Ind 클래스를 이용하여 이를 표현한다.

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

STGM의 제어흐름은 대부분 테일 호출(tail call) 형태이다. 역시 기존의 목적 기계와 달리 자바가상기계는 테일 호출을 위한 명령어를 제공하지 않는다. 따라서, 어느 정도 수행 부담을 감수하고 다음의 루프를 통해 이를 구현한다. 이 루프는 런 타임 시스템을 표현하는 클래스의 메소드로 포함된다. 처음 프로그램이 시작될 때 loopflag에 true를 할당하여 루프를 반복 수행하다 L-code 명령 STOP에 의해 loopflag는 false로 바뀌고 루프 수행이 멈춘다.

```
public static void loop(Clo cloptr) {
    while(loopflag) cloptr = cloptr.self.code();
}
```

런 타임 시스템에 필요한 구성요소는 노드 레지스터, 태그 레지스터, 스택과 관련 포인터들이 있다. 이들은 다음의 런 타임 클래스 G의 필드들로 표현한다.

```
public class G {
    public static Object node;
    public static int tag;
    public static boolean loopflag;
    public static int sp, bp;
    public static Object[] stk;
        ...
}
```

5. L-code-to-Java 컴파일러

본 절에서는 호이스팅 변환을 거친 L-code를 받아 자바 프로그램을 만드는 컴파일러를 설명한다. 앞서 설명했듯이 이 L-code는 단일 let 블록으로 내부에 또 다른 let을 포함하지 않는 L-code로만 구성되어 있다. 각 L-code 명령어를 자바 코드로 바꾸는 규칙은 [1]에 제시되어 있다.

컴파일러의 기본 원칙은 L-code의 각 클로져(let 바인딩, $l = \{\bar{x}\}.C$)마다 하나의 클래스를 선언한 다음 환경 $\{\bar{x}\}$에 해당하는 클래스 필드를 만들고, C에 해당하는 자바 코드를 만들어 메소드 code()를 만든다. 다음은 l_h 바인딩에 대해 만들어진 자바 클래스이다.

```
class Ch extends Clo {                          // class h
    Object f1, f2;                              // g, x
    Clo code() {
        Object z = G.node;                      // GETV
        Object g = this.f1;                     // GETC
        Object x = this.f2;
        int s = G.bp;                           // SAVE
        Cupd o = new Cupd();                    // PUTK
        o.f1 = z;
        o.f2 = s;
        G.sp++; G.sp=G.stk[G.sp] = o;
```

```
        G.bp = G.sp;                          // DUMP
        G.sp++; G.sp=G.stk[G.sp] = x;         // PUTA
        return (Clo) g;                       // JMPC
    }
}
```

6. 구현 및 실험

본 논문에서 제안한 컴파일 방법을 5개의 작은 Haskell 프로그램에 적용해 수행 성능을 측정하였다. Glasgow Haskell Compiler(GHC) 4.04를 사용해 Haskell 프로그램을 STG 프로그램으로 변환하였다. 이 때 GHC에 구현되어 있는 최적화 변환을 그대로 적용할 수 있다. 이 STG 프로그램을 받아 앞서 제시한 두 단계 과정을 거쳐 자바 프로그램을 생성하는 컴파일러를 구현하였다.

생성된 자바 프로그램은 SUN JIT 컴파일러(v.1.2.2)로 UltraSPARC-II 워크스테이션에서 실행하였다. 비교 대상은 GHC 4.04를 통해 얻은 바이너리 실행파일의 수행 성능과 Hugs98 (Feb-2000)에서 수행 성능이다. 먼저, 아래의 표는 컴파일 결과 얻은 GHC의 바이너리 실행파일 크기와 클래스 파일의 크기와 개수를 보여준다.

프로그램	GHC	JIT	JIT(최적화)
fib	268K	25K(37개)	19K(34개)
edigits	283K	114K(135개)	64K(92개)
prime	280K	74K(96개)	50K(70개)
queen	275K	109K(134개)	75K(101개)
soda	303K	336K(388개)	186K(203개)

다음의 표는 GHC, Hugs, 최적화 하지 않은 자바 프로그램, 최적화한 자바 프로그램을 실행해 얻은 수행 시간이다.

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

실험결과 생성된 자바 프로그램으로부터 얻어진 클래스 파일의 개수는 작은 크기의 Haskell 프로그램임에도 그 수가 많은 편이다. 특히 soda의 경우 바이트코드 크기가 GHC에서 생성한 바이너리보다 크다. 클래스 파일의 개수와 크기를 줄이는 문제를 고려할 필요가 있다.

실험 결과 생성된 자바 프로그램의 수행 시간은 GHC의 최적화 변환을 적용한 경우 GHC와 Hugs에서의 수행 시간 사이 정도에 위치하고 있음을 알 수 있다. soda의 경우 자바가상기계가 시작하는데 걸리는 시간(대략 0.68s~0.74s)으로 인해 사실상 Hugs에서의 수행시간인 0.41s 보다 빠를 수 없다.

7. 토의

STGM을 사용한 Tullsen과 Vernet의 경우 벤치마킹한 결과를 제시하지 않아서 비교할 수 없었다. G-Machine을 기반으로 컴파일러를 만든 Meehan&Joy[8]과 <ν,G>-Machine을 기반으로 한 Wakeling[3]의 경우 각각 벤치마킹 결과를 제시하였다. Meehan&Joy의 경우 벤치마크 프로그램이 본 논문에서 사용한 것과 동일하여 대강의 비교를 할 수 있는데 그 논문에서는 생성된 자바 프로그램의 수행시간이 Hugs에서 보다 5~10배 정도 느리다고 발표하였다. Wakeling의 경우 크기가 큰 벤치마크 프로그램을 사용하여 보다 실제적인 상황에서의 수행시간을 측정했다. Wakeling의 논문에서는 생성된 자바 프로그램이 Hugs에서의 수행시간 정도의 성능을 낸다고 발표하였다. 비록 본 논문에서 사용한 벤치마크 프로그램이 작아서 앞으로 더 큰 벤치마크 프로그램을 통한 실험이 필요하다.

클래스 파일의 개수가 많은 이유는 클로져마다 하나의 클래스를 생성하는 방식 때문이다. Wakeling은 자바가상기계에서 동적으로 클래스를 로딩하므로 클래스 파일의 개수가 수행 시간에 상당한 영향을 끼친다고 주장하고 클래스를 줄이는 방법을 한가지 제시하였다. 이 방법은 본 논문에서의 컴파일 방법에도 적용가능한데, 생성된 클래스들 중 동일한 타입과 개수의 필드들을 갖는 클래스를 하나의 클래스로 합하는 방법이다. 즉, 동일한 타입과 개수의 환경을 갖는 여러 클로져들을 동일한 클래스로 표현하는 것이다. 사실 본 논문에는 언급하지 않았지만 이 방법을 이미 구현하였고, 그 결과 생성된 클래스 개수는 20개 안팎으로 줄었고 바이트코드 크기는 반 이상이 줄었다.

한가지 주목할 사실은 클래스 파일 개수를 줄이도록 구현하기 위해서 STG-to-L-code 컴파일러는 거의 수정하지 않았고 L-code-to-Java 컴파일러만을 수정한 점이다. 이는 첫 번째 컴파일러에서 의도하는 STGM을 구현하는 작업과 연관되는 것이 아니라 두 번째 컴파일러에서 의도하는 자바로 표현하는 작업과 연관되기 때문이다.

L-code 프로그램에서 제어흐름 정보가 주어지면 이를 이용해 클로져 수준에서 인라이닝 변환을 적용해 테일 호출을 위한 인터프리터 루프 수행 부담을 줄이거나 클로져 개수(클래스 개수)를 줄일 수 있을 것이다. 이러한 분석과 변환은 첫 번째 컴파일러와 연관되는 작업이다.

8. 결론

본 논문에서는 자바가상기계를 목적 대상으로 하는 지연계산기반 함수형 언어 컴파일러를 설계하고 구현하였다. 기존 컴파일러들과 달리 STGM을 수행 모델로 삼았고, STGM에 대한 최적화 변환을 이용할 수 있었다. 또한 STGM을 구현하는 부분과 이를 자바로 표현하는 부분으로 모듈화하여 설계했다. 첫째 컴파일러를 간단한게 구체적인 규칙으로 명시할 수 있고, 둘째 특정 부분에 관련된 작업시 다른 부분에 대해 거의 고려하지 않아도 되는 장점을 지닌다.

9. 참고 문헌

[1] K. Choi, H. Lim, and T. Han. Compiling Lazy Functional Programs for the Java Virtual Machine on the Basis of the Spineless Tagless G-Machine. KAIST Technical Report, August 2000.

[2] A. Vernet. The Jaskell Project. A diploma project, February 1998.

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

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

[5] M. Wand. Correctness of Procedure Representations in Higher-Order Assembly Language. In S. Brooks, editor, Proceedings Mathematical Foundations of Programming Semantics '91, volume 598 of Lecture Notes in Computer Science, pages 294-311. Springer-Verlag, 1992.

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

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

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

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