스택 머신 코드의 타입 분석*
(progress report)

이욱세 이광근†
한국과학기술원 전산학과

요 약

본 논문에서는 스택 머신 코드의 타입을 분석해야 할 이유를 제기하고, 요약해석(abstract interpretation) 방법론에 따라 안전한 분석 방법을 설계하고, 그 구현한 결과를 보인다. 이 문제는 우리가 EM 코드를 PowerPC 어셈블리어 코드로 변환하는 컴파일러를 개발하는데 드러난 문제로써, 스택의 타입 정보가 있으면 보다 효율적인 코드를 생성하는데 사용할 수 있기 때문이다. 실행 중에 발생할 수 있는 모든 스택의 타입을 계산하는 것은 불가능하므로 요약해석 방법론을 적용한다. 요약한 스택 타입을 정의하고 이를 이용하여 스택 타입을 계산하는 방법을 제시한다. 제시한 분석 방법을 적용한 결과 5–10% 정도 속도가 향상된 코드를 생성하게 되었다. 분석 방법의 안전성은 현재 엄밀하게 증명하는 중이다.

1 배경

우리는 EM 코드[TvSKS83]를 PowerPC 어셈블리어 코드[Kac95]로 변환하는 컴파일러를 개발하였는데, 핵심이 되는 스택 머신 코드의 타입 분석에 대해 설명하려 한다.

LG 정보통신에서는 CHILL 언어로 작성된 스위칭 소프트웨어를, PowerPC 스위칭 시스템에서 사용하기 위해, PowerPC 용 컴파일러가 필요했다. 기존의 CHILL 컴파일러는, 중간언어인 스택 머신을 기반으로 하는 EM 코드로 컴파일한 후, 해당 기계로 컴파일하는 두 단계를 거치는데, 추가적으로 EM-to-PowerPC 컴파일러의 개발이 필요했던 것이다.

SML/NJ[MA93] 후단부에 사용된 MLRISC 프레임웍[Geo97]을 사용하여, EM-to-PowerPC 컴파일러를 개발하였다. 장점은, 첫째, 개발 과정을 MLTree를 중간 언어로 하여 두 과정으로 나누었기 때문에, 불필요한 과정의 연계를 막는다는 것이다. 중간 언어 MLTree는 레지스터를 기반으로 하기 때문에 PowerPC 코드와 크게 의미가 다르지 않고, 게다가 레지스터가 무한하다는 가정이 있으므로 레지스터 할당과 같은 후반부에서 생각할 것을 전반부와 연계시키지 않을 수 있다는 것이다. 둘째, SML/NJ 후단부에 사용된 최적화 코드를 사용할 수 있다는 것이다. SML/NJ는 널리 쓰이는 컴파일러로, 사용된 코드들은 사용자들에 의해 그 안전성이 검사되어진 것들이다. 마지막으로, 추후 MLRISC 프레임웍이 지원하는 다른 어셈블리어로의 컴파일러 개발이 용이하다는 것이다. EM 코드를 중간 언어 MLTree로 변환했기 때문에, MLRISC 프레임웍이 지원하는 기계들의 코드로 변환하는 컴파일러를 쉽게 만들 수 있다.

EM 코드를 MLTree로 변환하는데 있어, 실행 시간 스택의 타입 정보가 있으면 보다 효율적인 코드를 생성할 수 있음을 알아 내었다. 기본 블럭(basic block) 사이에 스택을 매개체로 값을 전달하는 경우, 스택의 타입 정보가 있으면 레지스터를 통해 값을 전달하는 코드를 생성할 수 있음을 알게 되었다.

본 논문에서는 스택 머신 코드의 타입을 분석하는 방법을 제시하고, 설계된 분석 방법이 안전하게 끝난다는 것을 보이며, 그 구현 결과를 보인다. 실행 중에 발생할 수 있는 모든 스택의 타입을 계산하는 것은 불가능하므로, 요약해석 방법론[CC77, CC92]을 적용하여, 요약한 스택 타입을 정의하고, 이를 이용하여 스택 타입을 계산하는 방법을 설명한다. 또한, 실험 결과로 5–10% 정도 속도가 향상된 코드를 생성하게 되었음을 보인다.

*본 연구는 LG 정보통신의 지원을 받았음을 밝힙니다.
†E-mail: {cookcu; kwang}@cs.kaist.ac.kr, web: http://cs.kaist.ac.kr/~{cookcu; kwang}

$$
\begin{array}{lcl}
\mathit{label} & \in & \mathit{String} \\
\tau & := & i \mid f \\
\mathit{size} & \in & \mathbf{N} \\
v_i & \in & \mathbf{Z} \\
v_f & \in & \mathbf{R} \\
n & \in & \mathbf{N}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{sprog} & ::= & (\mathit{label}\text{:} \mid \mathit{instr})^* \\
\mathit{instr} & ::= & \mathit{sop}_\tau \\
 & \mid & \texttt{CMP}_\tau\ \mathit{boolop} \\
 & \mid & \texttt{PUSH}\ v_\tau \\
 & \mid & \texttt{LOAD}\ \mathit{size} \\
 & \mid & \texttt{STORE}\ \mathit{size} \\
 & \mid & \texttt{POP}\ \mathit{size} \\
 & \mid & \texttt{DUP}_\tau \\
 & \mid & \texttt{JMP}\ \mathit{label} \\
 & \mid & \texttt{BCC}\ \mathit{label} \\
\mathit{sop} & ::= & \texttt{ADD} \mid \texttt{SUB} \mid \texttt{MUL} \mid \texttt{DIV} \\
\mathit{boolop} & ::= & = \mid < \mid > \mid \neq \mid \leq \mid \geq
\end{array}
$$

(a) 입력 언어

$$
\begin{array}{lcl}
\mathit{tprog} & ::= & (\mathit{label}\text{:} \mid \mathit{stmt})^* \\
\mathit{stmt} & ::= & r_n^\tau \leftarrow e_\tau \\
 & \mid & \texttt{M}[e_i] \leftarrow e_\tau \\
 & \mid & \texttt{PUSH}\ e_\tau \\
 & \mid & \texttt{JMP}\ \mathit{label} \\
 & \mid & \texttt{BCC}\ e_b\ \mathit{label} \\
e_\tau & ::= & v_\tau \\
 & \mid & r_n^\tau \\
 & \mid & \texttt{M}_\tau[e_i] \\
 & \mid & e_\tau\ \mathit{top}\ e_\tau \\
 & \mid & \texttt{POP}_\tau \\
e_b & ::= & e_\tau\ \mathit{boolop}\ e_\tau \\
\mathit{top} & ::= & + \mid - \mid * \mid /
\end{array}
$$

(b) 결과 언어

그림 1: 입력 언어와 결과 언어

2 언어

문제를 정확히 기술하기 위하여, 입력 언어(source language)와 결과 언어(target language)를 정의하자. 그림 1에 입력 언어와 결과 언어가 기술되어 있다. 실제 개발된 컴파일러의 입력 언어와 결과 언어에 비해서는 매우 단순하지만, 문제를 기술하고 설명하는데 있어서 충분하다. 입력 언어는 EM 코드의 주요 명령어들을 추려 낸 언어이고, 결과 언어는 레지스터를 기반으로 하는 중간 언어 MLTree를 추려 낸 언어이다. 앞서 말했다시피, 결과 언어를 중간 언어로 한 것은 실제 기계어에서 생길 수 있는 문제들을 제외하여, 논문의 초점이 흐려지지 않도록 하기 위해서이다.

입력 언어는 스택 기반 언어이다. 스택은 한 단어(word)를 기본 단위로 한다. 사용되는 값은 정수형과 실수형이 있는데, 정수형의 크기는 한 단어이고, 실수형의 크기는 두 단어이다. 실수형 값을 스택에 쌓거나 뺄 때, 스택의 두 단어의 공간이 사용된다. "PUSH v"는 값 v를 스택에 쌓는 명령어이다. "POP w"는 w 단어를 스택에서 제거한다. "sop_τ"는 스택에서 τ형의 값 두 개를 가져와서 계산한 후 그 결과를 스택에 쌓는다. "LOAD w"는 스택에서 정수형 번지 값을 가져와서 그 번지 값이 지정하는 메모리로부터 w 단어 크기의 값을 읽어 스택에 쌓는다. "STORE w"는 스택에서 번지 값과 w 단어 크기의 값을 읽어 번지 값이 지정하는 메모리에 쓴다. "$\texttt{CMP}_\tau$"는 두 τ 형 값을 스택으로부터 가져와서 비교한 후 참이면 1을 거짓이면 0을 스택에 쌓는다. "$\texttt{DUP}_\tau$"는 스택 위의 τ형 값을 한 번 더 쌓는다. "JMP *label*"는 무조건 분기하고, "BCC *label*"는 조건 분기로 스택에서 정수값을 가져와 0이 아니면 분기하고 0이면 분기하지 않는다.

결과 언어는 레지스터 기반 언어이다. 레지스터는 무한히 많다고 가정하고, 정수형 레지스터와 실수형 레지스터가 확실히 구분되어 사용된다고 가정한다. r_n^τ은 τ형 레지스터를 의미하고, n은 여러 개의 레지스터를 구분하기 위한 인덱스이다. $\texttt{M}[e_i]$는 e_i가 가르키는 주소의 메모리를 의미한다. PUSH와 $\texttt{POP}_\tau$는 실행 시간 스택 명령어로, 문제 설명에 필요하여 삽입하였다. 예를 들어, 스택 포인터를 r_1^i라 한다면, PUSH e_i는 다음 두 명령어와 동일한 일을 한다: $r_1^i \leftarrow r_1^i + 1$와 $\texttt{M}[r_1^i] \leftarrow e_i$. 다른 명령어들은 쉽게 이해될 수 있을 것이다.

입력 코드	결과 코드 A	결과 코드 B	결과 코드 C
PUSH 30	PUSH 30	$r_1^i \leftarrow 30$	
PUSH 10	PUSH 10	$r_2^i \leftarrow 10$	
PUSH 20	PUSH 20	$r_3^i \leftarrow 20$	
$\texttt{ADD}_i$	PUSH $(\texttt{POP}_i + \texttt{POP}_i)$	$r_4^i \leftarrow r_3^i + r_2^i$	$\texttt{M[0]} \leftarrow (20+10)-30$
$\texttt{SUB}_i$	PUSH $(\texttt{POP}_i - \texttt{POP}_i)$	$r_5^i \leftarrow r_4^i - r_1^i$	
PUSH 0	PUSH 0	$r_6^i \leftarrow 0$	
STORE 1	$\texttt{M[POP}_i\texttt{]} \leftarrow \texttt{POP}_i$	$\texttt{M}[r_6^i] \leftarrow r_5^i$	

그림 2: 코드 생성 예

입력 코드	코드 스택	출력된 결과 코드
PUSH 30	[30]	
PUSH 10	[10,30]	
PUSH 20	[20,10,30]	
$\texttt{ADD}_i$	[20+10,30]	
$\texttt{SUB}_i$	[(20+10)−30]	
PUSH 0	[0,(20+10)−30]	
STORE 1	[]	M[0]←(20+10)−30

그림 3: 컴파일 시간 스택 시뮬레이션 방법

3 문제

문제는 스택 기반 언어를 효율적인 레지스터 기반 언어로 변환하는 것이다. 스택, 즉, 메모리가 아닌 레지스터를 임시 저장 장소로 사용하고, 생성되는 기계어 코드의 수가 적을수록 효율적인 코드라 할 수 있다. 예제를 살펴 보면서 어떠한 코드가 효율적인가 살펴 보자.

스택을 사용하는 코드보다는 레지스터를 사용하는 코드가 효율적이다. 그림 2의 코드 A와 코드 B를 비교해 보자. 코드 A는 실행 시간 스택 명령어를 사용하여 생성된 코드이다. 코드 B는 코드 A에서 임시 저장 장소로 쓰이고 있는 스택 대신 레지스터를 사용하도록 한 코드이다. 코드 A는 메모리를 최소 13 번 접근하고, 코드 B는 최소 한 번 접근하므로, 코드 B가 더 효율적인 코드이다.

코드 B와 코드 C를 비교해 보면, 코드 C가 더 최적화된 기계어 코드로 생성될 가능성이 높다. 예를 들어, 기계어 코드가 "mv r,e", "add r,e,e", "sub r,e,e", "store e,r"으로 구성되어 있다고 하자. 여기서 r은 레지스터를 의미하고, e는 레지스터 또는 상수를 의미한다. mv은 e의 값을 r에 넣는 명령어, add는 e들을 더하여 r에 넣는 명령어, sub는 빼는 명령어, store는 r 값을 e 번지의 메모리에 저장하는 명령어이다. 코드 B는 7개의 기계어 코드로 생성되나, 코드 C는 다음과 같이 3개의 기계어 코드로 생성될 수 있다: add r_1,10,20; sub r_2,r_1,30; store 0,r_2. 즉, 임시 저장 장소를 쓰지 않고 보다 큰 계산식으로 생성되었을 때 효율적인 기계어 코드 생성이 가능한 것이다.

컴파일 시간 스택 시뮬레이션 방법을 사용하면, 코드 C와 같은 효율적인 코드를 생성해 낼 수 있다. 방법은, 컴파일 시간에 코드 스택(code stack)을 사용하여 실행 시간에 스택이 변하는 모습을 시뮬레이션 하면서 코드를 생성해 내는 것이다. 코드 스택은 그림 1.(b)의 e_τ, e_b를 원소로 하는 스택이다. $[e_1, e_2, \cdots, e_n]$와 같이 코드 스택을 쓸 때, 왼쪽, 즉 e_1을 스택 입구(top)로 본다. 만약, 코드 스택이 $[e_1, e_2, e_3]$일 때 $\texttt{ADD}_i$ 명령어가 입력되면 코드 스택은 $[e_1 + e_2, e_3]$으로 변한다. 다음 입력이 STORE 4 이면 $\texttt{M}[e_1 + e_2] \leftarrow e_3$ 코드를 생성하고 코드 스택은 비게 된다. 그림 3의 예제를 보면 코드 스택 방법을 쉽게 이해할 수 있을 것이다.

그러나, 코드 스택 방법은 기본 블럭(basic block) 내에서만 효율적인 코드를 생성할 수 있다. 즉, 분기 명령어에 대해서는 효율적인 코드를 생성해 낼 수 없다. 그림 4를 보자. 프로그램은

```
A:  BCC C          A:  BCC e_b C
B:  PUSH 50        B:  r_1^i ←50
    PUSH 30            r_2^i ←30
    JMP D              JMP D
C:  PUSH 40        C:  r_1^i ←40
    PUSH 20            r_2^i ←20
D:  ADD_i          D:  ··· ← r_2^i + r_1^i
```

(a) 입력 코드 (b) 결과 코드

그림 4: 분기 명령어에서의 코드 생성

4 개의 기본 블럭으로 이루어져 있고, 가능한 제어 흐름은 A→B→D와 A→C→D이다. 초기 코드 스택은 어떤 불린(boolean) 코드 e_b이다. 기본 블럭 A가 입력으로 주어지면 코드 스택은 비게 된다. B가 주어지면 $[30, 50]$이 되고, C가 주어지면 $[20, 40]$이 된다. D를 입력으로 받기 전의 코드 스택은, 두 가지 경우, B를 실행 한 후와 C를 실행한 후의 경우가 있을 수 있다. 그러나, $[30, 50]$과 $[20, 40]$을 동시에 나타낼 수 있는 코드 스택은 없다. 그러므로, B, C의 코드 뒤에, 코드 스택에 쌓인 코드들을 실제 스택에 쌓는 명령어를 삽입하고, D의 코드 이전에 실제 스택에 있는 것들을 코드 스택에 쌓아 주는 방법으로, 값을 전달해야 한다.

문제는 기본 블럭 사이에 값을 전달할 때 임시 저장 장소로 무엇을 사용할 것인가이다. 코드 스택 방법만으로는 기본 블럭 간에 코드 스택에 쌓여 있는 것들을 레지스터로 옮길 수 없다. 왜냐하면, 스택에 몇 개의 원소가 쌓여 있는지 알 수 없으므로, 코드 생성이 난해하다. 게다가, PowerPC 의 경우, 정수형 레지스터와 실수형 레지스터가 확연히 구분되어 있고, 두 레지스터 사이에 값의 전달이 용이하지 않으므로, 값의 타입을 알지 못하면 레지스터를 사용하여 값을 전달할 수가 없다.

스택에 쌓이는 원소들의 수와 타입을 안다면, 값을 전달하는 임시 저장 장소로 레지스터를 사용할 수 있다. 그림 4의 경우, B와 C가 끝난 뒤, 그리고, D가 실행되기 전에 스택에 두 개의 정수형 값이 쌓여 있다는 것을 확인할 수 있다. 그러면, 두 개의 임시 레지스터 r_1^i, r_2^i를 사용하여 기본 블럭 간에 값을 전달할 수 있다. 즉, B, C가 끝났을 때 코드 스택의 코드들을 r_1^i, r_2^i에 저장하는 코드를 생성하고, D 직전의 코드 스택을 $[r_2^i, r_1^i]$라 두고 컴파일하면 되는 것이다. 그림 4.(b)가 생성된 효율적인 코드이다.

4 스택의 타입 정보 분석

스택의 타입 정보를 분석하는 방법을 설계하기 위하여, 우선, 관심이 되는 스택의 내용(stack configuration)을 기술하는 도메인(domain)을 정의한다. 일반적으로 모든 가능한 스택의 내용을 계산하는 것은 불가능하므로, 이를 계산가능 하도록 요약한 도메인(abstract domain)을 정의한다. 요약된 도메인 내의 필요한 연산들, 특히 합치는 연산(join operator)을 정의하고, 이들을 이용하여 분석 함수를 설계하면, 요약 해석 방법론에 의해 스택의 타입을 유추할 수 있는 것이다.

4.1 스택 내용 도메인

관심이 되는 실행 시간의 스택의 내용을 기술하는 도메인은, 다음과 같이 원소들의 타입들의 리스트로 정의할 수 있다.

$$\boxed{\begin{array}{c} S := B \mid I \mid F \mid U_w \mid N \\ D := S^* \end{array}}$$

입력 언어를 보면 정수(I), 실수(F)의 두 가지 타입을 볼 수 있다. 불린 타입(B)은 결과 언어에서 구분되므로, 정수 타입과 구분지어 줄 필요가 있다. 정해지지 않은 타입(U_w)은 아직 그 원소

$$\|\epsilon\| = 0$$
$$\|B\| = \|I\| = \|N\| = 1$$
$$\|F\| = 2$$
$$\|U_w\| = w$$
$$\|xy\| = \|x\| + \|y\|, \text{ for } \forall x, y \in S^*$$

(a) 스택의 크기

$$\text{for all } x, y \in S^*,\ x \cdot y = xy$$

(b) 붙이는 연산

for all $s \in S$ and $x \in D$,

$$hd_0 x = \epsilon \qquad tl_0 x = x$$
$$hd_d s = s, \text{ if } \|s\| = d \qquad tl_d s = \epsilon, \text{ if } \|s\| = d$$
$$hd_1 F = N \qquad tl_1 F = N$$
$$hd_d U_w = U_d, \text{ if } d < w \qquad tl_d U_w = U_{w-d}, \text{ if } d < w$$
$$hd_d(sx) = \begin{cases} hd_d s, & \text{if } d \le \|s\| \\ s \cdot hd_{d-\|s\|} x, & \text{otherwise} \end{cases} \qquad tl_d(sx) = \begin{cases} tl_d s \cdot x, & \text{if } d \le \|s\| \\ tl_{d-\|s\|} x, & \text{otherwise} \end{cases}$$

(c) 자르는 연산

그림 5: 스택 내용의 기본 연산

가 특정 타입으로 사용되지 않았음을 의미한다. 예를 들어 "LOAD w"는 스택으로부터 정수 타입 메모리 번지를 읽고, 그 곳으로부터 자료를 가져와 스택에 쌓는다. 여기서 쌓인 자료는 특정한 타입으로 사용되지 않았다. 이러한 타입을 정해지지 않은 타입이라 하고 w 워드 크기의 정해지지 않은 타입을 U_w라 쓴다. 정의할 수 없는 타입(N)은 특정 타입으로 결정할 수 없는 타입을 말한다. 예를 들어, 정수 타입으로도 쓰이고 실수 타입으로도 쓰이면, 특정 타입의 레지스터에 저장할 수가 없기 때문에, 특정 타입으로 결정할 수 없는 타입 N이 되는 것이다.

스택 내용에 대한 기본적인 연산을 그림 5와 같이 정의한다. 스택 내용의 크기와, 두 스택 내용을 붙이는 연산(concatenation), 그리고, 스택 내용을 자르는 연산이다. 자르는 연산은 앞에서(hd) 그리고 뒤에서(tl) 자르는 연산이 필요하다. 실수 타입 F는 크기가 2 워드인데, 한 워드씩 자르면 N이 됨을 주목하라. 즉, $hd_1 F = N$ 이고 $tl_1 F = N$이다. 이는 실수 타입이 나뉘어져 사용되면 타입을 정의할 수 없기 때문이다.

관심은 모든 프로그램 지점에서의 가능한 스택의 내용이나, 이러한 가능한 스택의 내용을 일반적으로 정확히 계산할 수 없다.

$$\boxed{D^\natural := 2^D}$$

예를 들어, 한 번 반복할 때 정수 타입 하나를 스택에 쌓는 반복문이 있다고 하면, 초기 스택 내용이 $\{\epsilon\}$라 할 때, 반복문이 끝나는 지점에서 가능한 스택 내용은 $\{\epsilon, I, II, III, IIII, \cdots\}$이다. 이를 계산하면 무한히 반복되어 끝나지 않는다.

4.2 요약 도메인

적어도 스택 입구에서부터 가능한 만큼 정확한 원소들의 타입을 알아내는 것이 중요하다. 예를 들어, II, III가 가능한 스택 내용이라고 한다면, 스택 입구서부터 두 개의 원소는 정수 타입이 다라는 사실을 알아 내고 싶은 것이다.

$$
\begin{array}{l}
\text{for all } x, y \in S^*, \\
x \cdot^\sharp \bot = \bot \cdot^\sharp x = \bot \\
x \cdot^\sharp \top = x\top \text{ and } \top \cdot^\sharp x = \top \\
x \cdot^\sharp y = xy
\end{array}
$$

(a) 붙이는 연산자

$$
\begin{array}{l}
\text{for all } s \in S \text{ and } x \in D, \\
hd^\sharp_d \bot = \bot \qquad\qquad tl^\sharp_d \bot = \bot \\
hd^\sharp_d \top = \overbrace{N \cdots N}^{d} \qquad\qquad tl^\sharp_d \top = \top \\
hd^\sharp_d x = \begin{cases} hd_d x, & \text{if } d \le \|x\| \\ x, & \text{otherwise} \end{cases} \qquad tl^\sharp_d x = \begin{cases} tl_d x, & \text{if } d \le \|x\| \\ \bot, & \text{otherwise} \end{cases} \\
hd^\sharp_d (x\top) = \begin{cases} hd_d x, & \text{if } d \le \|x\| \\ x \cdot^\sharp hd^\sharp_{d-\|x\|} \top, & \text{otherwise} \end{cases} \qquad tl^\sharp_d (x\top) = \begin{cases} tl_d x \cdot^\sharp \top, & \text{if } d \le \|x\| \\ \top, & \text{otherwise} \end{cases}
\end{array}
$$

(b) 자르는 연산

$$
\begin{array}{l}
\bot \sqsubseteq x \text{ and } x \sqsubseteq \top, \text{ for } \forall x \in D^\sharp \\
U_1 \sqsubseteq B \sqsubseteq I \sqsubseteq N \text{ and } U_2 \sqsubseteq F \\
\text{For all } x, y \in D^\sharp \text{ and } s, t \in S, \\
sx \sqsubseteq ty, \text{ if } \begin{cases} \|s\| \le \|t\|, \ s \sqsubseteq hd^\sharp_{\|s\|} t, \text{ and } x \sqsubseteq tl^\sharp_{\|s\|}(ty) \\ \qquad\qquad \text{or} \\ \|s\| > \|t\|, hd^\sharp_{\|t\|} s \sqsubseteq t, \text{ and } tl^\sharp_{\|t\|}(sx) \sqsubseteq y \end{cases}
\end{array}
$$

(c) 스택 타입의 순서

그림 6: 스택 타입의 기본 연산과 순서

다음과 같은 요약 도메인은 그러한 정보를 담을 수 있는데, 이를 *스택 타입(stack type)*이라 하겠다.

$$\boxed{D^\sharp := \bot \mid S^* \mid S^*\top}$$

단, S^*의 스택 크기는 유한한 것으로 제한한다. $\top$은 모든 것이 가능함을 의미한다. 예를 들어, $II\top$은 스택 입구부터 두 개의 원소는 정수 타입이고, 그 밑은 모든 것이 가능하므로 모르겠다를 뜻한다. $\bot$은 가능한 스택의 내용이 없다를 의미한다. 어떤 프로그램 지점이 수행될 수 없으면 가능한 스택의 내용이 없다.

스택 내용의 기본 연산들을 확장하여 스택 타입의 기본 연산들을 정의하고, 이들을 토대로 분석의 핵심인 합치는 연산을 정의할 수 있다. 합치는 연산은 두 개 이상의 스택 내용이 가능할 때, 두 스택 내용을 함의하는 또 다른 스택 타입을 결정하는 연산이다. 즉, 어떤 프로그램 지점에서 스택 타입 A, B가 모두 가능하다면, 그 지점의 스택 타입은 $A \sqcup B$이 되는 것이다. 합치는 연산은 순서(order)의 정의에 의해 정의될 수 있다. $A \sqcup B$는 $A \sqsupseteq C$, $B \sqsupseteq C$ 를 만족하는 C 중 가장 순서가 낮은 것이 된다. 그림 6에 스택 타입의 기본 연산들과 순서가 정의되어 있다. 스택 타입 순서의 정의를 살펴 보면, 같은 크기의 스택 타입 원소들 사이에 순서가 존재하고 원소들의 리스트인 스택 타입 사이에는 크기별도 나누어서 순서가 존재하면 순서가 있게 됨을 알 수 있다. 예를 들어, $IIF\top \sqcup FFF\top$은 앞에서부터 크기에 따라 합쳐 보면 $NNF\top$이 됨을 알 수 있다.

$$type(e) = \text{case } e \text{ of} \begin{cases} op_\tau, & \tau\tau \to \tau \\ \texttt{CMP}_\tau\ boolop, & \tau\tau \to B \\ \texttt{PUSH}_\tau\ v_\tau, & \epsilon \to \tau \\ \texttt{LOAD}\ w, & I \to U_w \\ \texttt{STORE}\ w, & IU_w \to \epsilon \\ \texttt{POP}\ w, & U_w \to \epsilon \\ \texttt{DUP}_\tau, & \tau \to \tau\tau \\ \texttt{JMP}\ label, & \epsilon \to \epsilon \\ \texttt{BCC}\ label, & B \to \epsilon \end{cases}$$

$$F_f^\sharp(e,s) = \sigma_2 \cdot tl_{\|\sigma_1\|}s \text{ where } type(e) = \sigma_1 \to \sigma_2$$
$$F_b^\sharp(e,s) = \sigma_1 \cdot tl_{\|\sigma_2\|}s \text{ where } type(e) = \sigma_1 \to \sigma_2$$

그림 7: 타입 분석 함수

4.3 타입 분석

명령어로부터 타입 정보를 얻을 수 있는데, 이를 *명령어 타입(instruction type)*이라고 하겠다. 예를 들어, $\texttt{ADD}_i$는 정수 타입 두 개를 사용하여 정수 타입 한 개를 쌓는다. 이를 $II \to I$라고 쓸 수 있다. 이러한 명령어 타입은 입력 언어의 의미 구조(semantics)로부터 자동으로 유추될 수 있다. 입력 언어의 명령어 타입이 그림 7에 있다.

타입 정보는 명령어의 수행 전후로 전달되기 때문에, 앞쪽으로 가면서 분석하는 것과 뒤쪽으로 가면서 분석하는 것이 모두 필요하다. 예를 들어, "`PUSH 10; PUSH 20; STORE 1`"와 같은 프로그램에서 `STORE 1` 직전의 스택 타입은 $II\cdots$ 이다. 이는 앞의 `PUSH 10` 명령어로부터 `STORE 1`에서 저장할 값이 정수 타입이라는 정보를 준 것이다. "`LOAD 1; DUP`i"와 같은 프로그램에서는, `LOAD 1`에서 읽어온 값이 정수 타입이라는 정보를 뒤의 $\texttt{DUP}i$가 주고 있다.

명령어의 타입으로부터 앞쪽부터 또는 뒤쪽부터 분석하는 계산식이 그림 7의 $F_f^\sharp$, $F_b^\sharp$ 이다. 두 계산식은 각각 명령어 e가 수행되기 전 또는 수행된 후의 스택 타입 σ를 주면 수행된 후 또는 수행전의 스택 타입을 결과로 준다. 앞쪽부터 분석하는 경우, 명령어가 스택으로부터 읽어 오는 만큼 자른 뒤, 명령어가 스택에 쌓는 원소들의 타입들을 붙여 주는 것이다.

정의된 타입 분석 함수를 사용하여 요약 해석 방법론[CC77, CC92]에 의해 분석한 예제가 그림 8에 있다. 프로그램은 0부터 50까지의 정수를 `M[`*interval*`]` 간격으로 증가시키면서 스택에 쌓는 프로그램이다. 초기 스택 타입을 II라 하면, 앞쪽부터 분석하는 방법만으로 두 번 반복한 뒤에 분석이 끝나게 된다. 분석된 결과로부터 `START` 블럭과 `LOOP` 블럭에서 `LOOP` 블럭으로 스택 입구로부터 세 개의 원소를 정수형 레지스터를 사용하여 값을 전달할 수 있음을, `LOOP` 블럭에서 `EXIT` 블럭으로 네 개의 원소를 정수형 레지스터를 사용하여 값을 전달할 수 있음을 알게 되었다.

4.4 분석의 안전성

제시한 타입 분석은 안전하다. 안전하다는 것은 실제 일어날 스택의 내용이 분석 결과에 포함되어 있다는 것이다. 예를 들면, 어떤 프로그램 지점에서 $II\top$이 분석 결과로 주어지면, 실제 실행 시에 I와 같이 $II\top$이 포함할 수 없는 ($I \not\sqsubseteq II\top$) 내용을 스택이 가지지 않는다는 것이다.

정리 1 (안전성). *제시된 타입 분석을 통해 얻은 분석 결과는 수행 중에 발생하는 스택의 내용을 포함한다.*

우선, $D^\natural$와 $D^\sharp$의 관계를 설명하는, 요약 함수(abstraction function) α와 해석 함수(concretization function) δ를 제시하고, 이를 이용하여, 두 도메인이 갈로아 관계(Galois connection)임을 밝

	1번 반복	2번 반복	마지막 반복
START:	II		
PUSH 0	III		
		$III \sqcup IIII =$	$III\top \sqcup IIII\top =$
LOOP:	III	$III\top$	$III\top$
$\mathtt{DUP}_i$	$IIII$	$IIII\top$	
PUSH *interval*	$IIIII$	$IIIII\top$	
LOAD 1	U_1IIII	$U_1IIII\top$	
$\mathtt{ADD}_i$	$IIII$	$IIII\top$	
$\mathtt{DUP}_i$	$IIIII$	$IIIII\top$	
PUSH 50	$IIIIII$	$IIIIII\top$	
$\mathtt{CMP}_i$ =	$BIIII$	$BIIII\top$	
BCC LOOP	$IIII$	$IIII\top$	
EXIT:			

그림 8: 타입 분석 예

힌다. 그리고, $D^\sharp$에서의 입력 언어 의미 구조(collecting semantic)를 $F_f^\sharp$, $F_b^\sharp$라 한다면, 모든 $s \in D^\sharp$에 대해 $F_f^\sharp(s) \sqsupseteq (\alpha \circ F_f^\sharp \circ \gamma)(s)$와 $F_b^\sharp(s) \sqsupseteq (\alpha \circ F_b^\sharp \circ \gamma)(s)$가 성립함을 밝히면 된다.

제시된 타입 분석은 유한 시간 내에 완료된다.

정리 2 (유한성). *제시된 타입 분석은 항상 유한 시간 내에 완료된다.*

요약 도메인 $D^\sharp$에 무한 체인(infinite chain)이 없다는 것과, 분석 함수 $F_f^\sharp$, $F_b^\sharp$ 가 단조(monotone) 함수임을 증명하면 된다.

5 실험 결과

타입 분석 정보를 이용한 것과 이용하지 않은 컴파일러가 생성한 코드 사이에서는 5–10%정도의 속도 차이를 보였다. 그림 9가 실험 결과이다. 비교 대상이 된 컴파일러는 자체 제작한 EM-to-PowerPC 컴파일러를 타입 분석 정보를 이용한 버전과 이용하지 않은 버전으로 다르게 만든 컴파일러이다. 컴파일러가 수행된 환경은 Sun UltraSparc 296 Mhz 기계이고, 컴파일된 코드가 수행된 환경은 PowerPC 604e 166 Mhz 가 장착된 IBM RS6000 이다. 실제 개발된 컴파일러의 목적인 스위칭 소프트웨어 소스들은, 프로그램들이 실행될 수 있는 PowerPC를 기반으로 한 병렬 스위칭 시스템을 LGIC 측에서 아직 제공하지 못해, 실행 시간은 조사하지 못하였고 결과 코드의 크기만 비교하였다.

실험 결과에서 "PL_onl_ld"의 경우 결과 코드가 커졌는데, 이는 경험적인(heuristic) 레지스터 할당 루틴[GA96]과 연계되어 악효과를 준 것으로 예상된다. 레지스터 할당 루틴은, 어느 순간 레지스터가 시스템이 제공하는 수보다 많이 사용되는 경우, 특정 레지스터를 희생자로 골라 임시 메모리로 바꾼다. 즉, 그 레지스터를 사용하는 명령어들을 메모리 사용 명령어들로 변환하여, 사용 레지스터 수를 줄이는 것이다. 이 때, 메모리에서 읽고 쓰는 과정이 필요하므로, 명령어의 수는 증가하게 된다. 타입을 분석하여 변환한 결과 코드는, 타입 분석을 하지 않고 변환한 결과 코드가 사용한 레지스터 외에, 추가의 레지스터를 사용하게 된다. 문제는 추가된 레지스터 때문에 필요로 하는 레지스터의 수가 시스템이 제공하는 수보다 많아진 경우이다. 추가된 레지스터를 희생자로 하면, 분석한 것과 분석하지 않은 것의 결과 코드가 유사한 크기의 코드를 생성하게 된다. 왜냐하면, 분석하지 않은 경우에도 스택이라는 메모리에 읽고 쓰는 코드가 생성되기 때문이다. 그러나, 추가된 것이 아닌, 보다 빈번히 사용된 다른 레지스터를 희생자로 하는 경우, 명령어 수가 많아지게 되는 것이다.

입력 코드	입력 코드 크기 (line)	분석 시간 (ms)[a]	타입 분석 없이 변환한 코드		타입을 분석하여 변환한 코드	
			크기 (line)	실행시간 (ms)[b]	크기 (line)	실행시간 (ms)[b]
factor	73	1	104	15.4	88	14.7
8_queen	279	18	363	22.3	348	20.1
quick_sort	222	14	279	93.4	263	91.9
MPDH_main[c]	1751	193	4471	-	4435	-
PL_onl_ld[c]	1742	392	1675	-	1774	-
SSL_crox[c]	4114	530	20412	-	20286	-

[a]Sun Ultra Sparc 296MHz

[b]PowerPC 604e 166 Mhz

[c]병렬 스위칭 소프트웨어 소스들로, 실행 환경인 PowerPC 기반 병렬 스위칭 시스템이 제공되지 못하여, 실행시간을 조사하지 못하였다.

그림 9: 실험 결과

6 결론

본 논문에서는 스택 머신 코드의 타입을 분석하는 방법을 제시하고, 그 분석의 안전성을 보였다. 실제 현장에서 사용되는 컴파일러 개발에 있어, 분석이 필요한 부분에서 이론적으로 잘 정리되어 있는 요약해석 방법론을 적용하여 안전한 분석 방법을 제시하고 구현하였다. 그 결과 5–10% 정도 실행 속도가 향상된 코드를 생성하는 EM-to-PowerPC 컴파일러를 개발하였다. 현재, 분석 방법의 안전함을 엄밀하게 증명하고 있는 중이다.

참고 자료

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic Computation*, 2(4):511–547, 1992. Also as a tech report: Ecole Polytechnique, no. LIX/RR/92/10.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3), 1996.

[Geo97] Lal George. MLRISC: Customizable and reusable code generators. Lucent Bell Laboratories, 1997.

[Kac95] Gary Kacmarcik. *Optimizing PowerPC^{TM} code.* Addison-Wesley, 1995.

[MA93] David B. MacQueen and Andrew W. Appel. Standard ML of New Jersey. Technical memorandum, AT&T Bell Labs., 1993.

[TvSKS83] Andrew S. Tanenbaum, Hans van Staveren, Ed G. Kaizer, and Johan W. Stevenson. Description of a machine architecture for use with block structured languages. Technical report, Vrije Universiteit, Amsterdam, 1983.