스택 머신 코드의 타입 분석*

이욱세 이광근 김병철 권경인
한국과학기술원 전산학과 LG 정보통신

Reconstructing the Types of Stack-Machine Codes

Oukseh Lee, Kwangkeun Yi, Byungchul Kim, and Kyoung-In Kwon
Dept. of Computer Science, KAIST LG Information & Communications, Ltd.

요 약

스택 머신 코드(stack-machine code)를 레지스터 기반 언어로 변환하는데 있어 스택의 타입 정보가 있으면 보다 효율적인 코드를 생성할 수 있음을 알아 내었다. 본 논문에서는 스택 머신 코드의 타입을 분석해야 할 이유를 제기하고, 요약해석 방법론에 따라 분석 방법을 제시하고, 제시된 분석 방법의 안전성을 짚어 본다.

1 배경

우리는 EM 코드[6]를 PowerPC 어셈블리어 코드[5]로 변환하는 컴파일러를 개발하였는데, 핵심이 되는 스택 머신 코드의 타입 분석에 대해 설명하려 한다.

LG 정보통신*에서는 CHILL로 작성된 스위칭 소프트웨어를, PowerPC 스위칭 시스템에서 사용하기 위해, PowerPC용 컴파일러가 필요했다. 기존의 CHILL 컴파일러는, 중간언어인 EM 코드로 컴파일한 후, 해당 기계로 컴파일하는 두 단계를 거치는데, 추가적으로 EM-to-PowerPC 컴파일러의 개발이 필요했던 것이다.

컴파일러 개발에 SML/NJ[1] 후단부에 사용된 MLRISC 프레임웍[4]를 사용하였다. 장점은, EM 코드를 MLRISC 프레임웍의 중간 표현인 MLTree로 변환하는 과정과 PowerPC 명령어로 변환하는 과정을 나누어 생각할 수 있다는 것, SML/NJ 후단부에 사용된 최적화 코드를 사용할 수 있다는 것, 그리고, MLRISC 프레임웍이 지원하는 다른 어셈블리어로 컴파일이 가능하다는 것이다.

스택 머신 코드(stack-machine code)인 EM을 레지스터 기반 언어인 MLTree로 변환하는데 있어 스택의 타입 정보가 있으면 보다 효율적인 코드를 생성할 수 있음을 알아 내었다.

본 논문에서는 스택 머신 코드의 타입을 분석하는 방법을 제시한다. 실행 중에 발생할 수 있는 모든 스택의 타입을 계산하는 것은 불가능하므로, 요약해석(abstract interpretation) 방법론[2, 3]을 적용하여, 요약한 스택 타입을 정의하고, 이를 이용하여 모든 프로그램 지점의 스택 타입을 계산하는 방법을 설명한다.

2 언어

문제를 정확히 기술하기 위하여, 입력 언어와 결과 언어를 정의하자. 그림 2에 입력 언어와 결과 언어가 기술되어 있다.

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 번지 값(정수형)을 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.

$$
\begin{array}{lcl}
label & \in & String \\
\tau & := & i \mid f \\
size & \in & \mathbf{N} \\
v_i & \in & \mathbf{Z} \\
v_f & \in & \mathbf{R} \\
n & \in & \mathbf{N} \\
sprog & ::= & (label\!: \mid instr)^* \\
instr & ::= & sop_\tau \\
 & \mid & \mathtt{CMP}_\tau\ boolop \\
 & \mid & \mathtt{PUSH}\ v_\tau \\
 & \mid & \mathtt{LOAD}\ size \\
 & \mid & \mathtt{STORE}\ size \\
 & \mid & \mathtt{POP}\ size \\
 & \mid & \mathtt{DUP}_\tau \\
 & \mid & \mathtt{JMP}\ label \\
 & \mid & \mathtt{BCC}\ label \\
sop & ::= & \mathtt{ADD} \mid \mathtt{SUB} \mid \mathtt{MUL} \mid \mathtt{DIV} \\
boolop & ::= & = \mid < \mid > \\
 & & \mid \neq \mid \leq \mid \geq
\end{array}
$$

(a) 입력 언어

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

(b) 결과 언어

그림 1: 입력 언어와 결과 언어

입력 코드	결과 코드 A	결과 코드 B
PUSH 30	PUSH 30	$r^i_1 \leftarrow 30$
PUSH 10	PUSH 10	$r^i_2 \leftarrow 10$
PUSH 20	PUSH 20	$r^i_3 \leftarrow 20$
$\mathtt{ADD}_i$	PUSH $(\mathtt{POP}_i + \mathtt{POP}_i)$	$r^i_4 \leftarrow r^i_3 + r^i_2$
$\mathtt{SUB}_i$	PUSH $(\mathtt{POP}_i - \mathtt{POP}_i)$	$r^i_5 \leftarrow r^i_4 - r^i_1$
PUSH 0	PUSH 0	$r^i_6 \leftarrow 0$
STORE 1	$\mathtt{M}[\mathtt{POP}_i] \leftarrow \mathtt{POP}_i$	$\mathtt{M}[r^i_6] \leftarrow r^i_5$

그림 2: 코드 생성 예

"$\mathtt{CMP}_\tau$"는 두 τ 형 값을 스택으로부터 가져와서 비교한 후 참이면 1을 거짓이면 0을 스택에 쌓는다. "$\mathtt{DUP}_\tau$"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^τ_n은 τ형 레지스터를 의미하고, n은 여러 개의 레지스터를 구분하기 위한 인덱스이다. $\mathtt{M}[e_i]$는 e_i가 가르키는 주소의 메모리를 의미한다. PUSH와 $\mathtt{POP}_\tau$는 실행 시간 스택 흉내 명령어이다. 다른 명령어들은 쉽게 이해될 수 있을 것이다.

3 문제

문제는 스택 기반 언어를 효율적인 레지스터 기반 언어로 변환하는 것이다.

*본 연구는 LG 정보통신의 지원을 받았음을 밝힙니다.

source code	code stack	emitted target code
`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

```
A:  BCC C
B:  PUSH 50
    PUSH 30
    JMP D
C:  PUSH 40
    PUSH 20
D:  ADDi
```

(a) 입력 코드

```
A:  BCC eb C
B:  r1i ←50
    r2i ←30
    JMP D
C:  r1i ←40
    r2i ←20
D:  ··· ← r2i + r1i
```

(c) 결과 코드

그림 4: 분기 명령어에서의 코드 생성

스택을 사용하는 코드보다는 레지스터를 사용하는 코드가 효율적이다. 그림 3의 코드 A와 코드 B를 비교해 보자. 코드 A는 메모리를 최소 13 번 접근하고, 코드 B는 최소 한 번 접근하므로, 코드 B가 더 효율적인 코드이다.

코드 B와 결과 코드 C (`M[0]` $\leftarrow (20+10)-30$) 를 비교해 보면, 코드 C가 더 최적화된 기계어 코드로 생성될 가능성이 높다. 예를 들어, 기계어 코드가 "$r \leftarrow r+e$", "$r \leftarrow e$", "$r \leftarrow r-e$", "$M[e] \leftarrow e$"으로 구성되어 있다고 하자. 여기서 r은 레지스터를 의미하고, e는 레지스터 또는 상수를 의미한다. 코드 B는 7개의 기계어 코드로 생성되나, 코드 C는 다음과 같이 3개의 기계어 코드로 생성될 수 있다: $r_1 \leftarrow 10+20$, $r_2 \leftarrow r_1 - 30$, $M[0] \leftarrow r_2$. 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. 코드 스택(code stack)은 그림 2.(b)의 e_i, e_f, e_b를 원소로 하는 스택이고, $[e_1, e_2, \cdots, e_n]$와 같이 쓴다. 왼쪽, 즉 e_1을 스택 입구(top)로 본다. 만약, 코드 스택이 $[e_1, e_2, \cdots, e_n]$일 때, $\texttt{ADD}_i$ 명령어가 입력되면, $[e_1+e_2, e_3]$으로 변한다. 다음 입력이 `STORE 4` 이면 $M[e_1+e_2] \leftarrow e_3$ 코드를 생성하고, 코드 스택은 비게 된다. 코드 스택 방법은, 실행 시간에 스택이 변하는 모습을 코드 스택으로 흉내를 내어, 코드를 생성해 내는 것이다. 그림 3의 예제를 보면 쉽게 이해할 수 있을 것이다.

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

문제는 기본 블럭 사이에 값을 전달해야 할 때, 임시 저장 장소로 무엇을 사용할 것인가이다.

스택에 쌓이는 원소들의 개수와 타입(type)을 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, B, C가 끝났을 때 코드 스택의 코드를 r_1^i, r_2^i에 저장하는 코드를

$$
\begin{aligned}
&\|\epsilon\| = 0\\
&\|B\| = \|I\| = \|N\| = 1\\
&\|F\| = 2\\
&\|U_w\| = w\\
&\|xy\| = \|x\| + \|y\|,\ \text{for } \forall x, y \in \widehat{\mathbf{S}}*
\end{aligned}
$$

(a) 크기

$$
\begin{aligned}
&\text{for all } x, y \in \widehat{\mathbf{S}}^*,\\
&x \cdot \bot = \bot \cdot x = \bot\\
&x \cdot \top = x\top \text{ and } \top \cdot x = \top\\
&x \cdot y = xy
\end{aligned}
$$

(b) 붙이는 연산자(concatenation)

$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{for all } s \in \widehat{\mathbf{S}} \text{ and } x \in \widehat{\mathbf{S}}^*,}\\
hd_d \bot = \bot & tl_d \bot = \bot\\
hd_0 x = \epsilon & tl_0 x = x\\
hd_d \top = \overbrace{N \cdots N}^{d} & tl_d \top = \top\\
hd_d x = x,\ \text{if } \|x\| = d & tl_d x = \epsilon,\ \text{if } \|x\| = d\\
hd_1 F = N & tl_1 F = N\\
hd_d U_w = U_d,\ \text{if } d < w & tl_d U_w = U_{w-d},\ \text{if } d < w\\
hd_d(sx) = & tl_d(sx) =\\
\quad\begin{cases} hd_d s, & \text{if } d \le \|s\| \\ s \cdot hd_{d-\|s\|} r, & \text{otherwise} \end{cases} & \quad\begin{cases} tl_d s \cdot x, & \text{if } d \le \|s\| \\ tl_{d-\|s\|} x, & \text{otherwise} \end{cases}
\end{array}
$$

(c) 자르는 연산자

그림 5: 스택 타입의 기본 연산자

생성하고, D 직전의 코드 스택을 $[r_2^i, r_1^i]$라 두고 컴파일하면, 레지스터를 통해서 기본 블럭 간에 값을 전달할 수 있다. 그림 4.(c)가 생성된 효율적인 코드이다.

4 스택 타입(Stack Type)

스택의 내용을 원소들의 타입들의 리스트라 하자.

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

입력 언어를 보면 4 가지 원소의 타입을 볼 수 있다: 정수(I), 실수(F), 불린(b), 정해지지 않은 타입(U). 불린 타입은 결과 언어에서 구분되므로, 정수 타입과 구분지어 줄 필요가 있다. 정해지지 않은 타입은 아직 그 원소가 특정 타입으로 사용되지 않았음을 의미한다.

관심은 모든 프로그램 지점에서의 가능한 스택의 내용이다.

$$
\boxed{\mathbf{D} := 2^{\mathbf{C}}}
$$

그러나, 이러한 가능한 스택의 내용을 일반적으로 정확히 계산할 수 없다.

그렇기 때문에, 가능한 스택의 내용을 요약해야 한다. 적어도 스택 입구에서부터 가능한 만큼 정확한 원소들의 타입을 알아내는 것이 중요하다. 다음과 같은 도메인(domain)은 이러한 정보를 담을 수 있다. 이를 스택 *타입(stack type)*이라 하겠다.

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

$\top$은 모든 것이 가능함을 의미한다. $\bot$은 가능한 스택의 내용이 없다를 의미한다. 어떤 프로그램 지점이 수행될 수 없으면 가능한 스택의 내용이 없다. N은 정의할 수 없는 타입을 의미한다. 정수 타입으로도 쓰이고 실수 타입으로도 쓰이면, 특정 타입의 레지스터에 저장할 수가 없다.

$$\perp \sqsubseteq x \text{ and } x \sqsubseteq \top,\ \text{for } \forall x \in \widehat{\mathbf{D}}$$
$$U_1 \sqsubseteq B \sqsubseteq I \sqsubseteq N \text{ and } U_2 \sqsubseteq F$$
$$\text{For all } x, y \in \widehat{\mathbf{D}} \text{ and } s, t \in \widehat{\mathbf{S}},$$
$$sx \sqsubseteq ty,\ \text{if} \begin{cases} \|s\| \leq \|t\|,\ s \sqsubseteq hd_{\|s\|}t,\ \text{and } x \sqsubseteq tl_{\|s\|}(ty) \\ \text{or} \\ \|s\| > \|t\|, hd_{\|t\|}s \sqsubseteq t,\ \text{and } tl_{\|t\|}(sx) \sqsubseteq y \end{cases}$$

그림 6: 스택 타입의 순서

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

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

그림 7: 타입 분석 함수

모든 프로그램 지점의 스택 타입을 계산하기 위하여 스택 타입에 대한 기본적인 연산자가 필요하다. 스택 타입 연산의 기본 단위가 크기이므로 스택 타입의 크기를 그림 5.(a)에 정의하였다. 두 스택 타입을 붙이는 연산(concatenation)(그림 5.(b))과, 자르는 연산(그림 5.(c))이 필요하다. 자르는 연산은 앞에서(hd) 그리고 뒤에서(tl) 자르는 연산이 필요하다.

기본적인 스택 타입 연산자들로부터 합치는 연산(join)을 정의할 수 있다. 합치는 연산은 두 개 이상의 스택 내용이 가능할 때, 두 스택 내용의 공통 부분을 표현하는 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. 스택 타입의 순서의 정의는 그림 6와 같다.

5 타입 분석(Type Reconstruction)

명령어로부터 타입 정보를 얻을 수 있다. 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 를어, "`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 `DUP`i가 주고 있다.

명령어의 타입으로부터 앞쪽부터 또는 뒤쪽부터 분석하는 계산식이 나올 수 있다. 두 계산식은 각각 명령어 e가 수행되기 전 또는 수행된 후의 스택 타입 σ를 주면 수행된 후 또는 수행전의 스택 타입을 결과로 준다.

제시한 타입 분석은 안전하다. 안전하다는 것은 실제 일어날 스택의 내용이 분석 결과에 포함되어 있다는 것이다.

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

제시된 타입 분석은 유한 시간 내에 완료된다. 요약 도메인 $\widehat{\mathbf{D}}$에 무한 체인(infinite chain)이 없다는 것과, 분석 함수

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

그림 8: 타입 분석 예

입력 코드	줄 수	분석 시간[a]	분석 없이		타입 분석	
			줄수	시간[b]	줄수	시간[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	1751	193	4471	-[c]	4435	-
PL_onl_ld	1742	392	1675	-	1774	-
SSL_crox	4114	530	20412	-	20286	-

[a]Sun Ultra Sparc 296MHz (ms)
[b]PowerPC 167MHz (ms)
[c]실제 스위칭 소프트웨어로 실행 환경이 아직 갖추어지지 않았다.

그림 9: 실험 결과

F_f, F_b 가 단조(monotone) 함수임을 증명하면 된다.

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

타입 분석의 예제가 그림 5에 있다. 프로그램은 0부터 50까지의 정수를 `M[`*interval*`]` 간격으로 증가시키면서 스택에 쌓는 프로그램이다. 초기 스택 타입을 II라 하고, 앞쪽부터 분석하는 방법만으로, 두 번 반복한 뒤에 분석이 끝나게 된다. 결과로 세 개의 정수값이 정수형 레지스터를 통해 `LOOP` 블럭에서 전달될 수 있음을 알 게 된다.

타입 분석 정보를 이용한 것과 이용하지 않은 컴파일러가 생성한 코드 사이에서는 5–10% 성도의 속도 향상을 보였다. 그림 5가 실험 결과이다. 비교 대상이 된 컴파일러는 자체 제작한 EM-to-PowerPC 컴파일러를 타입 분석 정보를 이용한 버전과 이용하지 않은 버전으로 다르게 만든 컴파일러이다.

6 결론

본 논문에서는 스택 머신 코드의 타입을 분석하는 방법을 제시하고, 그 분석의 안전성을 짚어 보았다. 실제 현장에서 사용되는 컴파일러 개발에 있어, 요약해석 방법론을 적용하여 구현하였다는데 의의가 있다고 본다.

참고 자료

[1] Andrew W. Appel and David B. MacQueen. Standard ML compiler. In *Lecture Notes in Computer Science*, volume 274. Springer-Verlag, 1987.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *4th ACM Symposium on Principles of Prog. Lang.*, pages 238–252, 1977.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Logic Computation*, 2(4):511–547, 1992.

[4] Lal George. MLRISC: Customizable and reusable code generators. 1997.

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

[6] 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.