(19) 대한민국특허청(KR)
(12) 공개특허공보(A)

(51) Int. Cl.[6]
G06F 9/45

(11) 공개번호 특2000-0056822
(43) 공개일자 2000년09월15일

(21) 출원번호 10-1999-0006512
(22) 출원일자 1999년02월26일

(71) 출원인 엘지정보통신 주식회사 서평원
서울특별시 강남구 역삼동 679
(72) 발명자 이광근
대전광역시유성구신성동한울아파트109동1501호
이욱세
충청북도청주시상당구수동243-2
김병철
서울특별시서초구방배3동1018-1번지삼익아파트3동708호
(74) 대리인 김영철

심사청구 : 있음

(54) 이엠 코드에 대한 실행시간 스택의 타입 분석 방법

요약

본 발명은 EM(End of Medium character) 코드에 대한 실행시간 스택의 타입 분석에 관한 것으로, 특히 스택을 기반으로 하는 EM 코드를 레지스터를 기반으로 하는 어셈블리어로 변환하는 컴파일러에 있어, 명령어 전/후에 실행시간 스택의 타입을 분석하도록 함으로써, 기본 블록간에 소정 값을 전달하는 경우 스택을 사용하지 않고 레지스터를 사용할 수 있도록 한 EM 코드에 대한 실행시간 스택의 타입 분석 방법에 관한 것이다.

종래에는 가상 스택 방법을 이용하여 컴파일을 수행하므로 기본 블록간에 스택을 통해 소정 값을 전달하는 경우 해당 전달 값들이 적재된 스택의 타입(적재된 값의 종류와 개수)을 알 수 없어 레지스터를 기반으로 하는 효율적인 코드를 생성할 수 없다는 문제점이 있고, 이에 따라 컴파일된 코드의 처리 속도가 늦은 단점이 있었다.

본 발명은 도메인과 타입 분석 함수 및 합치는 연산자를 정의하고, 해당 정의한 요소들을 이용하여 실행시간 스택의 타입을 분석함으로써, 기본 블록 내부만이 아니라 해당 기본 블록간에도 레지스터를 기반으로 하는 효율적인 코드를 생성할 수 있게 되어 컴파일된 코드의 처리 속도가 향상되는 효과가 있다.

대표도

도3

명세서

도면의 간단한 설명

도 1은 종래 컴파일러에 대한 입력언어와 결과언어를 정의한 도면.

도 2는 입력코드에 대해 생성된 결과코드를 도시한 도면.

도 3은 본 발명에 따른 실행시간 스택의 타입 분석을 적용한 컴파일러의 동작 순서도.

도 4는 본 발명에 따른 스택 타입의 기본 연산자와 합치는 연산자를 정의한 도면.

도 5는 본 발명에 따른 타입 분석 함수를 정의한 도면.

도 6은 본 발명에 따른 n!을 계산하는 입력코드에 대한 분석을 예시한 도면.

발명의 상세한 설명

발명의 목적

발명이 속하는 기술 및 그 분야의 종래기술

본 발명은 EM(End of Medium character) 코드에 대한 실행시간 스택의 타입 분석에 관한 것으로,

특히 스택을 기반으로 하는 EM 코드를 레지스터를 기반으로 하는 어셈블리어로 변환하는 컴파일러에 있어, 명령어 전/후에 실행시간 스택의 타입을 분석하도록 함으로써, 기본 블록간에 소정 값을 전달하는 경우 스택을 사용하지 않고 레지스터를 사용할 수 있도록 한 EM 코드에 대한 실행시간 스택의 타입 분석 방법에 관한 것이다.

일반적으로, 교환 시스템 프로그램은 실시간 동작을 중점적으로 수행해야 하므로 해당 시스템의 기능 수행에 있어서 실시간의 제한을 많이 받는다. 특히 하드웨어 장치를 제어하는 하위레벨 프로세스의 경우는 실시간 성격이 더욱 강하며, 상위레벨 프로세스의 경우는 하위레벨에 비하여 실시간 성격은 약하지만 하위레벨 프로세스에서 처리된 메시지의 처리 및 응답 등을 수행하는데 있어서 실시간 처리가 요구된다.

따라서, 대부분의 교환 시스템의 경우 실시간 성격이 강한 하위레벨 프로세스에서는 MC68xxx 어셈블리 언어와 C 언어를 사용하였으며, 보다 기능적 처리에 주된 기능을 수행하는 상위레벨 프로세스에서는 보다 쉬운 프로그래밍, 판독성(readability)의 향상, 소프트웨어의 유지보수성 향상, 이식성(portability)의 향상을 제공할 수 있는 CHILL(CCITT HIgh-Level Language) 언어를 사용하였다.

또한, 해당 CHILL 언어의 사용은 국제 경쟁력의 향상, 동시 처리(concurrent processing) 달성, 엄격한 타입 검사(type checking)에 의한 품질 향상 등을 제공할 수 있기 때문에 하위레벨 프로세스에서도 표준화를 위하여 CHILL 언어의 사용을 원칙으로 하고 있으며, 필요한 경우 어셈블리 명령어를 CHILL 언어와 혼용하여 사용하도록 권고하고 있다.

한편, CHILL 언어를 입력받아 시스템의 중앙처리장치에서 사용할 수 있는 실제 코드를 생성하는 컴파일러의 경우 ACK를 사용하여 개발할 수 있으며, 해당 ACK(Amsterdam Compiler Kit)에서는 컴파일러를 손쉽게 만들기 위해 어셈블리어 수준의 중간언어(intermediate language)인 EM 코드를 사용하는데, 해당 EM 코드는 스택을 기반으로 하는 언어로서, 함수 호출시 파라미터와 결과 값을 주고 받을 때, 산술 연산 등 기본적인 수행에 스택을 이용할 뿐, 레지스터를 다루는 명령어가 없고, 단지 내부적으로 쓰는 몇몇 레지스터들을 가정하고 있다.

그리고, 해당 EM 코드를 중간언어로 사용하는 컴파일러는 해당 CHILL 언어 즉, 소스(source) 언어를 입력받아 EM 코드로 변환하는 전단부(front-end) 컴파일러와, 해당 EM 코드를 실제 코드 즉, 목적(target) 어셈블리어로 변환하는 후단부(back-end) 컴파일러로 구성되는데, 이때 해당 EM 코드를 레지스터 기반으로 하는 기계어로 변환하는 후단부 컴파일러(현재, EM-to-SPARC 컴파일러가 구현되어 있다)에서는 가상 스택 방법을 사용하고 있다.

이러한 가상 스택 방법을 사용하는 컴파일러의 동작 원리를 첨부된 도면 도 1과 같이 입력언어와 결과언어를 정의하여 설명하면 다음과 같다.

해당 입력언어는 EM 코드의 주요 명령어들을 나타낸 언어이고, 해당 결과언어는 레지스터를 기반으로 하는 중간언어로서, 결과언어를 중간언어로 한 이유는 기술하고자 하는 종래 기술의 문제점을 보다 정확히 표현하기 위한 것으로 실제 기계어에서 발생될 수 있는 문제들 즉, 레지스터 할당(register allocation), 명령어 순서 배열(instruction scheduling) 등을 제외하였다.

여기서, 도 1의 (가)를 보다 상세히 설명하면, 해당 입력언어는 한 단어(word, 4바이트)를 기본 단위로 하는 스택을 기반으로 하는 언어로서, 사용되는 값으로는 크기가 한 단어인 정수형과, 크기가 두 단어 즉, 스택에 적재하거나 읽어낼 때 해당 스택의 두 단어 공간이 사용되는 실수형이 있다.

해당 명령어(instr)에는 스택에서 τ형의 값 두 개를 읽어들여 계산한 후 그 결과를 해당 스택에 적재하는 sop와, 스택에서 τ형의 값 두 개를 읽어들여 'boolop'에 따라 비교한 후 참이면 '1'을, 거짓이면 '0'을 해당 스택에 적재하는 CMP_τ boolop와, τ형의 값 $v(v_\tau)$를 스택에 적재하는 PUSH v_τ와, 스택에서 정수형 번지 값을 읽어들인 후, 해당 번지 값이 지정하는 메모리로부터 size 크기의 값을 읽어들여 해당 스택에 적재하는 LOI size와, 스택에서 정수형 번지 값과 size 크기의 값을 읽어들인 후, 해당 번지 값이 지정하는 메모리에 size 크기의 값을 기록하는 STI size와, size 크기의 값을 스택에서 제거하는 POP size와, 스택의 최상위에 적재된 τ형의 값을 한 번 더 적재하는 DUP_τ와, 무조건 label로 분기하는 JMP label과, 조건에 따라 즉, 스택에서 정수 값을 읽어들여 '0'이 아니면 분기하고, '0'이면 label로 분기하는 BCC label이 있다.

그리고, 도 1의 (나)를 보다 상세히 설명하면, 해당 결과언어는 정수형 또는 실수형 레지스터 기반의 언어로서, 해당 τ_n은 인덱스(index)가 n인 τ형 레지스터를 의미하고, $M[e_i]$는 e_i가 지시하는 주소의 메모리를 의미한다. 그리고, 해당 PUSH e_τ와, JMP label과, BCC e_b label과, POP_τ는 종래 기술의 문제점을 설명하기 위해 삽입한 스택 명령어로서, 예를 들어, 스택 포인터를 i_1이라 가정하면, PUSH e_i는 '$i_1 \leftarrow i_1+1$; $M[i_1] \leftarrow e_i$'의 두 명령어와 동일한 기능을 수행한다.

상술한 바와 같은 가상 스택 방법은 실행시간 스택(run-time stack)을 시뮬레이션하여 결과코드를 생성해 주는데, 해당 가상 스택은 결과코드(결과언어) 또는 이와 유사한 코드를 내용으로 하는 스택으로서, 도 1의 (나)에 도시된 e_τ, e_b를 원소로 하는 스택이며, [e_1, e_2, …, e_n]과 같이 가상 스택이 적재된 경우 왼쪽 즉, e_1을 스택 입구(top)로 본다.

또한, 해당 가상 스택 방법은 각각의 EM 명령어가 실행 중에 동작하는 것을 코드로 표현하되 실제 스택을 가상 스택으로 흉내를 내어 코드를 생성하는 것으로, 아래 표 1을 예를 들어 설명하면 다음과 같다.

[표 1]

입력코드	가상 스택	출력된 결과코드
PUSH b PUSH c ADD$_i$ SUB$_i$ PUSH 0 STI 1	[] [b] [c,b] [c+b] [i_0-(c+b)] [0,i_0-(c+b)] []	i_0←POP$_i$ M[0]←i_0-(c+b)

여기서, ADD$_i$ 명령에 의해 두 정수 코드인 'c'와 'b'를 가상 스택에서 읽어들여 가산한 코드 'c+b'가 가상 스택에 적재된 후, SUB$_i$ 명령을 수행해야 하는데, 이때, 해당 가상 스택에 하나의 값만이 적재되어 있으므로 실제 스택에서 읽어오는 명령을 생성하게 된다. 즉, 새로운 정수형 레지스터 'i_0'에 대해 'i_0←POP$_i$'를 생성하고 가상 스택은 [i_0]이 되어 해당 SUB$_i$ 명령을 수행한 결과, 해당 가상 스택에는 'i_0-(c+b)'가 적재된다. 그리고, PUSH 0 명령에 의해 '0,i_0-(c+b)'가 가상 스택에 적재된 후, STI 1 명령에 의해 결과코드 'M[0]←i_0-(c+b)'가 생성되고, 해당 가상 스택은 비게 된다.

그런데, 전술한 바와 같은 종래의 가상 스택 방법에서는 기본 블록(basic block) 내에서만 효율적인 코드를 생성할 수 있었다. 즉, 분기 명령어에 대해서는 효율적인 코드를 생성할 수 없었다. 여기서, 효율적인 코드라 함은 스택(메모리)이 아닌 레지스터를 임시 저장 장소로 사용하고, 생성되는 기계어 코드의 수가 적은 것 즉, 스택을 사용하는 코드보다는 레지스터를 사용하는 코드를 의미한다.

예를 들어, 첨부된 도면 도 3을 살펴보면 해당 EM 프로그램 즉, 입력코드 측은 4개의 기본 블록(A, B, C, D)으로 이루어져 있고, 가능한 제어 흐름은 'A→B→D'와 'A→C→D'가 있고, 초기 가상 스택은 소정의 불린(boolean) 코드 'e_b'이다. 여기서, 기본 블록 A가 입력으로 주어지면 가상 스택은 비게 되고, 기본 블록 B가 입력으로 주어지면 가상 스택은 [30,50]이 되며, 기본 블록 C가 입력으로 주어지면 가상 스택은 [20,40]이 된다. 따라서, 기본 블록 D를 입력으로 받기 전에 가상 스택은 두 가지의 경우 즉, 기본 블록 B를 실행한 후와 기본 블록 C를 실행한 후의 경우를 나타낼 수 있고, 해당 가상 스택 [30,50]과 [20,40]을 동시에 나타낼 수 있는 가상 스택은 없으므로, 종래에는 기본 블록 B, C의 입력코드를 수행한 후 해당 가상 스택에 적재된 코드들을 실제 스택에 적재하는 명령어를 삽입하고, 기본 블록 D의 입력코드 이전에 해당 가상 스택을 비우고 시작하면 해당 가상 스택을 통해 값이 전달되어 도 2와 같은 결과코드(1)가 생성된다.

따라서, 종래에는 가상 스택 방법을 이용하여 컴파일을 수행하므로 기본 블록간에 스택을 통해 소정 값을 전달하는 경우 해당 전달 값들이 적재된 스택의 타입(적재된 값의 종류와 개수)을 알 수 없어 레지스터를 기반으로 하는 효율적인 코드를 생성할 수 없다는 문제점이 있고, 이에 따라 컴파일된 코드의 처리 속도가 늦은 단점이 있었다.

발명이 이루고자하는 기술적 과제

본 발명은 전술한 바와 같은 문제점을 해결하기 위한 것으로 그 목적은, 도메인과 타입 분석 함수 및 합치는 연산자를 정의하고, 해당 정의한 요소들을 이용하여 실행시간 스택의 타입을 분석함으로써, 기본 블록 내부만이 아니라 해당 기본 블록간에도 레지스터를 기반으로 하는 효율적인 코드를 생성할 수 있도록 하는데 있다.

발명의 구성 및 작용

상기와 같은 목적을 달성하기 위한 본 발명의 특징은, 실행시간 스택을 기반으로 하는 EM 코드를 입력받아 레지스터를 기반으로 하는 어셈블리어 코드로 변환하는 컴파일러에 있어서, 분석 값을 표현하는 도메인과, 명령어에 대한 타입 분석 함수 및 분석 값을 합치는 연산자를 이용하여 상기 실행시간 스택의 타입을 분석한 후, 기본 블록간에 전달되는 분석 값을 저장할 레지스터를 할당하여 어셈블리어 코드를 생성하는데 있다.

한편, 상기 도메인은 스택의 내용이 없음을 표현하는 바텀 또는 원소들의 시퀀스 중 소정 길이 이하인 것들에 모든 것이 가능함을 의미하는 탑을 붙여 생성한 분석 값으로 표현하는 것을 특징으로 하며, 상기 타입 분석 함수는 EM 프로그램에 대해 앞에서부터 분석하는 순방향 분석 함수와, 해당 EM 프로그램에 대해 뒤에서부터 분석하는 역방향 분석 함수를 포함하는 것을 특징으로 하고, 상기 합치는 연산자는 두 개 이상의 스택 타입이 가능한 경우 해당 두 스택 타입을 함의하는 스택 타입을 결정하는 연산을 수행하도록 하는 것을 특징으로 한다.

본 발명의 다른 특징은, 실행시간 스택을 기반으로 하는 EM 코드를 입력받는 과정과; 입력받은 EM 코드에 대한 실행시간 스택의 타입을 합치는 연산자를 이용하여 타입 분석함수에 따라 분석한 분석 값을 소정의 도메인으로 표현하는 과정과; 상기 각 분석 값에 대한 도메인을 비교하여 변화하지 않는 경우 각 기본 블록 사이에 전달되는 분석 값을 저장할 레지스터를 할당하여 어셈블리어 코드를 생성하는 과정을 포함하는데 있다.

이하, 본 발명의 실시예를 첨부한 도면을 참조하여 상세하게 설명하면 다음과 같다.

본 발명에 따른 교환 시스템에서의 컴파일러는 첨부한 도면 도 3과 같은 과정을 통하여 컴파일하는데, 먼저, EM 프로그램이 입력되면, 입력된 EM 프로그램의 구문을 분석하여(스텝 S1), 내부 표현방식(abstract syntax tree)으로 변환한다.

그리고, 해당 EM 프로그램을 레지스터 기반의 중간언어인 MLTree로 변환하되, 해당 EM 프로그램의 분석 값을 표현하는 도메인(domain)과, 각 명령어에서의 타입 분석 함수 및 해당 분석 값을 합치는 연산자를 정의하여 실행시간 스택의 타입을 분석한 후(스텝 S2), 분석한 스택 타입 정보와 가상 스택 방법을 이용하여 EM 코드와 어셈블리어와의 의미적 차이(semantic gap)를 보완해주는 중간 표현인 해당 MLTree로 변환한다(스텝 S3).

이후, 변환한 MLTree를 레지스터가 무한히 많다는 가정하에 어셈블리어 코드로 생성하고(스텝 S4), 생성한 어셈블리어 코드에 대해 각각 레지스터를 할당한 후(스텝 S5), 해당 어셈블리어 코드를 출력함으로써, 해당 시스템의 프로세스에서 실시간 동작을 수행할 수 있게 된다.

특히, 본 발명에서는 교환 시스템의 컴파일 동작 중에 수행되는 EM 코드에 대한 실행시간 스택의 타입을 분석하는 방법을 제공함으로써, 소정의 기본 블록 내부에서만이 아니라 해당 기본 블록간에도 레지스터를 기반으로 하는 효율적인 코드를 생성할 수 있게 된다.

본 발명에 따른 EM 코드에 대한 실행시간 스택의 타입 분석은 분석 값을 표현하는 도메인과, 각 명령어에서의 타입 분석 함수 및 해당 분석 값을 합치는 연산자(join operation)를 정의한 후, 해당 도메인과 타입 분석 함수 및 합치는 연산자를 이용하여 일반적으로 널리 알려진 분석 틀인 "CC77"에 의해 분석할 수 있게 되는데, 이러한 동작은 해당 EM 코드가 입력으로 주어지는 경우 각 명령어 전/후에 실행시간 스택의 타입을 분석하게 되는 것이다.

해당 도메인(D)은 아래 수학식 1과 같이, 어떤 프로그램 지점이 수행될 수 없는 경우 가능한 스택의 내용이 없음을 표현하는 바텀(bottom) '⊥' 또는 'S'의 원소들의 시퀀스(sequence) 중 소정 길이(k) 이하인 것들에 모든 것이 가능함을 의미하는 탑(top ;⊤)이 붙어 이루어진 '$S^{\leq k}$⊤'로서 분석 값을 나타내게 되는데, 여기서, 해당 'S'는 불린(boolean) 타입의 원소(B) 또는 정수형 레지스터에 저장 가능한 원소(I) 또는 실수형 레지스터에 저장 가능한 원소(F) 또는 특정 타입으로 사용되지 않았음을 의미하는 정해지지 않은 타입의 원소(U_w) 또는 정의할 수 없는 타입 즉, 메모리를 통하여 연산을 수행해야 하는 타입의 원소(N)를 의미한다. 예를 들어, 'II⊤'은 스택 입구부터 두 개의 원소는 정수 타입이고, 그 이하는 모든 것이 가능하므로 아무 정보도 없음을 의미한다.

$$D = \perp \mid S^{\leq k}\top$$

해당 합치는 연산자는 두 개 이상의 스택 타입이 가능한 경우 첨부한 도면 도 4와 같이 몇 가지의 기본적인 연산자 즉, 스택 내용의 크기를 결정하는 연산자와, 앞 자르는 연산자(hd_d)와, 뒤 자르는 연산자(tl_d) 및 붙이는 연산자를 이용하여 해당 두 스택 타입을 함의(implication)하는 스택 타입을 결정하기 위한 연산자로서, 어떤 프로그램 지점에서 스택 타입 σ, τ가 모두 가능하다면, 그 지점의 스택 타입은 'σ∪τ'가 되는 것이다.

한편, 도 4의 (가)는 크기를 결정하는 연산자로서, 해당 크기 단위로는 단어(word)를 사용한다.

그리고, (나)는 앞 자르는 연산자(hd_d)로서, S^*의 원소를 입력으로 받아 스택의 앞으로부터 d 크기만큼 자른 값을 전달하는 연산을 수행하도록 하고, (다)는 뒤 자르는 연산자(tl_d)로서, S^*의 원소를 입력으로 받아 스택의 앞으로부터 d 크기만큼 자르고 남은 값을 전달하는 연산을 수행하도록 하는데, 만약, 입력이 d 크기보다 작은 경우 해당 입력 값을 전달하게 되지만, 실수 타입 F는 크기가 두 단어(2 word)이므로 한 단어씩 자르면 N이 된다. 즉, $hd_1F=N$이고 $tl_1F=N$이 되는데, 이것은 실수 타입이 나뉘어져 사용되면 타입을 정의할 수 없기 때문이다.

(라)는 두 스택 타입을 붙여주는 연산(concatenation)을 수행하도록 한 붙이는 연산자이며, (마)는 두 스택 내용의 공통 부분을 표현하는 스택 타입을 결정하는 연산(join)을 수행하도록 한 합치는 연산자로서, 합치는 연산은 순서(order) 정의에 의해 수행될 수 있는데, σ∪τ는 σ, τ≤υ를 만족하는 υ 중 가장 순서가 낮은 것이 되며, 이때, 해당 스택 타입의 순서는 같은 크기의 스택 타입 원소들 사이에 존재하고, 원소들의 리스트인 스택 타입 사이에는 크기별로 나누어서 존재한다.

해당 타입 분석 함수에는 첨부한 도면 도 5와 같이, EM 프로그램에 대해 앞에서부터 분석하는 순방향 분석 함수 '$F_e(s)$'와, 해당 EM 프로그램에 대해 뒤에서부터 분석하는 역방향 분석 함수 '$B_e(s)$'가 있는데, 본 발명에서는 EM 코드의 의미 구조(semantics)로부터 자동으로 유추할 수 있는 명령어 타입 정보 즉, 해당 명령어가 어떤 타입의 값들을 스택에서 읽어들여 어떤 타입의 값들을 스택에 적재하는가를 나타내는 정보가 해당 명령어의 수행 전/후로 전달되기 때문에 해당 순방향 분석 함수와 역방향 분석 함수를 혼합하여 분석하되, 해당 분석 결과가 변하지 않을 때까지 반복적으로 분석한다. 예를 들어, "PUSH 10; PUSH 20; STI 1"과 같은 프로그램에서 "STI 1" 수행 전의 스택 타입은 "II…"으로서, 이는 앞의 "PUSH 10" 명령어로부터 "STI 1"에서 저장할 값이 정수 타입이라는 정보를 전달한 것이며, "LOI 1; DUP_i"와 같은 프로그램에서는 "LOI 1"에서 읽어온 값이 정수 타입이라는 정보를 뒤의 "DUP_i"에서 전달한 것이다.

즉, 명령어 타입으로부터 전방향 또는 역방향으로 분석하는 계산식이 나올 수 있는데, 두 계산식은 각각 명령어 'e'가 수행되기 전 또는 수행된 후의 스택 타입 σ를 전달하면 수행되기 전 또는 수행된 후의 스택 타입을 결과로 전달해 주되, 명령어가 스택으로부터 읽어들인 만큼 자르는 연산자에 의해 자른 뒤, 해당 명령어가 스택에 적재하는 원소들의 타입들을 붙이는 연산자에 의해 붙

여주는 것이다.

이와 같은 실행시간 스택의 타입 분석 방법을 적용한 본 발명에 따른 컴파일러는 도메인과 타입 분석 함수 및 합치는 연산자를 이용하여 EM 코드 입력으로부터 스택 타입을 분석한 후, 기본 블록 사이의 분석 값에 따라 새로운 레지스터를 할당해 주고, 할당된 레지스터를 이용하여 가상 스택 방법에 따라 어셈블리어 코드를 생성하게 된다.

여기서, 해당 실행시간 스택의 타입 분석을 상세히 설명하면 다음과 같다.

먼저, EM 프로그램을 각 명령어 또는 기본 블록을 노드로 하는 제어 흐름 그래프(control-flow graph)로 나타내고, 각 노드에 분석 값을 저장할 수 있도록 한 후, 초기에는 모든 노드에 아무 것도 가능하지 않음을 나타내는 분석 값 '⊥'을 저장하되, 시작 노드에는 모든 것이 가능함을 나타내는 초기 값 '⊤'을 저장한다.

그리고, 각 노드에서 상위 노드의 분석 값을 타입 분석 함수에 입력으로 제공하여 연산된 결과 값을 저장하는데, 이때, 상위 노드가 다수 개일 경우 각 상위 노드들의 분석 값을 합치는 연산자를 이용하여 수행한 연산 결과 값을 입력으로 제공하여 모든 노드의 분석 값이 변화하지 않을 때까지 반복 수행함으로써, 변화하지 않는 각 노드의 분석 값 즉, 분석하고자 했던 결과 값을 얻게 된다.

이후, 분석이 종료되면 각 기본 블록 사이에 전달되는 분석 값을 저장할 레지스터를 할당하게 되는데, 예를 들어, 블록 A와 블록 B, C 사이의 분석 값이 'IIF⊤'인 경우 두 개의 정수 값과 한 개의 실수 값이 전달되는 것을 의미하므로, 사용되지 않은 새로운 정수형 레지스터 두 개와 실수형 레지스터 한 개를 할당하게 된다.

이렇게 할당된 레지스터를 이용하여 효율적인 코드를 생성할 수 있는데, 기본적으로 가상 스택 방법을 사용하여 코드를 생성하되, 기본 블록의 시작 지점과 끝 지점에서는 다른 방법으로 코드를 생성한다. 즉, 해당 기본 블록의 시작 지점에서는 기할당된 레지스터들이 가상 스택에 적재되어 있다고 보고 코드를 생성하며, 해당 기본 블록의 끝 지점에서는 남아있는 가상 스택의 코드들을 끝 지점에 할당된 레지스터로 옮기는 명령어를 삽입해 준다.

'n!'을 계산하는 입력 코드에 대한 분석을 첨부한 도면 도 6을 참조하여 설명하면 다음과 같다.

초기 스택 타입으로 시작점은 '⊤', 나머지는 '⊥'로 한 후, 순방향 분석 함수(F_e)로 앞에서부터 분석하고, 역방향 분석 함수(B_e)로 뒤에서부터 분석하면, 해당하는 분석 값이 생성되는데, 이때, 해당 타입 분석은 분석 값이 변하지 않을 때까지 반복적으로 수행하게 된다. 그리고, 해당 분석 값으로부터 레지스터를 할당하게 되는데, 해당 블록 S와 블록 A 또는 블록 B와 블록 A 사이의 분석 값 'II⊤'에 대해서는 레지스터 'i_0'와 'i_1'을 할당하고, 블록 A와 블록 B 또는 블록 A와 블록 C 사이의 'II⊤'에 대해서는 레지스터 'i_2'와 'i_3'을 할당하게 된다.

여기서, 블록 B를 컴파일하는 경우를 살펴보면, 초기 가상 스택을 할당된 레지스터 [i_2,i_3]으로 하여 가상 스택 방법에 의해 컴파일하고, 남은 가상 스택 [M_i[t],i_3*i_5]를 할당된 레지스터 'i_0'와 'i_1'에 저장하는 코드 즉, 'i_1←i_3*i_5; i_0←M_i[t];'와 같은 효율적인 코드를 생성하게 된다.

상술한 바와 같이, 본 발명에서는 가상 스택에 적재되는 원소들의 개수와 타입을 타입 분석 방법에 따라 알 수 있으므로 첨부된 도면 도 2와 같은 효율적인 결과코드(2)를 생성할 수 있게 된다. 즉, 분석 값을 전달하는 임시 저장장소로 레지스터를 사용할 수 있는데, 블록 B, C가 실행된 후, 블록 D가 실행되기 전에 스택에 두 개의 정수형 분석 값이 적재되어 있다는 것 즉, 스택의 타입 정보를 알 수 있어, 두 개의 새로운 레지스터 'i_1'와 'i_2'를 사용하여 기본 블록 사이에 분석 값을 전달할 수 있게 된다. 다시 말해서, 블록 B, C가 실행된 후, 가상 스택의 코드들을 레지스터 i_1'와 'i_2'에 저장하는 코드를 생성하고, 블록 D가 실행되기 전에 가상 스택을 [i_1,i_2]라 두고 컴파일하게 됨에 따라 해당 컴파일된 코드의 처리 속도가 향상된다.

발명의 효과

이상과 같이, 본 발명은 도메인과 타입 분석 함수 및 합치는 연산자를 정의하고, 해당 정의한 요소들을 이용하여 실행시간 스택의 타입을 분석함으로써, 기본 블록 내부만이 아니라 해당 기본 블록간에도 레지스터를 기반으로 하는 효율적인 코드를 생성할 수 있게 되어 컴파일된 코드의 처리 속도가 향상되는 효과가 있다.

(57) 청구의 범위

청구항 1

실행시간 스택을 기반으로 하는 EM 코드를 입력받아 레지스터를 기반으로 하는 어셈블리어 코드로 변환하는 컴파일러에 있어서,

분석 값을 표현하는 도메인과, 명령어에 대한 타입 분석 함수 및 분석 값을 합치는 연산자를 이용하여 상기 실행시간 스택의 타입을 분석한 후, 기본 블록간에 전달되는 분석 값을 저장할 레지스터를 할당하여 어셈블리어 코드를 생성하는 것을 특징으로 하는 이엠 코드에 대한 실행시간 스택의 타입 분석 방법.

청구항 2

제 1항에 있어서,

상기 도메인은, 스택의 내용이 없음을 나타내는 바텀 또는 원소들의 시퀀스 중 소정 길이 이하인 것들에 모든 것이 가능함을 의미하는 탑을 붙여 생성한 분석 값으로 표현하는 것을 특징으로 하는 이엠 코드에 대한 실행시간 스택의 타입 분석 방법.

청구항 3

제 2항에 있어서,

상기 원소는, 불린 타입의 원소 또는 정수형 레지스터에 저장 가능한 원소 또는 실수형 레지스터에 저장 가능한 원소 또는 특정 타입으로 사용되지 않았음을 의미하는 정해지지 않은 타입의 원소 또는 정의할 수 없는 타입의 원소인 것을 특징으로 하는 이엠 코드에 대한 실행시간 스택의 타입 분석 방법.

청구항 4

제 1항에 있어서,

상기 타입 분석 함수는, EM 프로그램에 대해 앞에서부터 분석하는 순방향 분석 함수와, 해당 EM 프로그램에 대해 뒤에서부터 분석하는 역방향 분석 함수를 포함하는 것을 특징으로 하는 이엠 코드에 대한 실행시간 스택의 타입 분석 방법.

청구항 5

제 1항에 있어서,

상기 합치는 연산자는, 두 개 이상의 스택 타입이 가능한 경우 해당 두 스택 타입을 함의하는 스택 타입을 결정하는 연산을 수행하도록 하는 것을 특징으로 하는 이엠 코드에 대한 실행시간 스택의 타입 분석 방법.

청구항 6

제 1항에 있어서,

상기 합치는 연산자는, 스택 내용의 크기를 결정하는 연산자와, 앞 자르는 연산자와, 뒤 자르는 연산자 및 붙이는 연산자를 포함하는 것을 특징으로 하는 이엠 코드에 대한 실행시간 스택의 타입 분석 방법.

청구항 7

제 6항에 있어서,

상기 앞 자르는 연산자는, 원소들을 입력으로 받아 스택의 앞으로부터 소정 크기만큼 자른 값을 전달하는 연산을 수행하도록 하는 것을 특징으로 하는 이엠 코드에 대한 실행시간 스택의 타입 분석 방법.

청구항 8

제 6항에 있어서,

상기 뒤 자르는 연산자는, 원소들을 입력으로 받아 스택의 앞으로부터 소정 크기만큼 자르고 남은 값을 전달하는 연산을 수행하도록 하는 것을 특징으로 하는 이엠 코드에 대한 실행시간 스택의 타입 분석 방법.

청구항 9

제 6항에 있어서,

상기 붙이는 연산자는, 두 개 이상의 스택 타입이 존재하는 경우 해당 스택 타입을 붙여주는 연산을 수행하도록 하는 것을 특징으로 하는 이엠 코드에 대한 실행시간 스택의 타입 분석 방법.

청구항 10

실행시간 스택을 기반으로 하는 EM 코드를 입력받는 과정과; 입력받은 EM 코드에 대한 실행시간 스택의 타입을 합치는 연산자를 이용하여 타입 분석함수에 따라 분석한 분석 값을 소정의 도메인으로 표현하는 과정과; 상기 각 분석 값에 대한 도메인을 비교하여 변화하지 않는 경우 각 기본 블록 사이에 전달되는 분석 값을 저장할 레지스터를 할당하여 어셈블리어 코드를 생성하는 과정을 포함하는 것을 특징으로 하는 이엠코드에 대한 실행시간 스택의 타입 분석 방법.

청구항 11

제 10항에 있어서,

상기 분석 값을 소정의 도메인으로 표현하는 과정은, 입력받은 EM 코드에 대한 실행시간 스택의 타입을 합치는 연산자를 이용하여 전방향 분석함수에 따라 분석한 전방향 분석 값을 소정의 도메인으로 표현하는 단계와; 입력받은 EM 코드에 대한 실행시간 스택의 타입을 합치는 연산자를 이용하여 후방향 분석함수에 따라 분석한 후방향 분석 값을 소정의 도메인으로 표현하는 단계를 더 포함하는 것을 특징으로 하는 이엠코드에 대한 실행시간 스택의 타입 분석 방법.

도면

도면1

(가)

$$
\begin{array}{lcl}
label & \in & String \\
\tau & ::= & i(integer) \mid f(float) \\
size & \in & \mathbf{N}\,(\text{자연수}) \\
v_i & \in & \mathbf{Z}\,(\text{정수}) \\
v_f & \in & \mathbf{R}\,(\text{실수}) \\
n & \in & \mathbf{N}
\end{array}
$$

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

(나)

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

도면2

입력코드

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

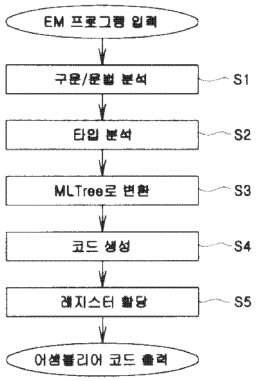
결과코드(1)

```
A: BCC eb C
B: PUSH 50
   PUSH 30
   JMP D
C: PUSH 40
   PUSH 20
D: i1 ← POPi
   i2 ← POPi
   ··· ← i1 + i2
```

결과코드(2)

```
A: BCC eb C
B: i1 ← 50
   i2 ← 30
   JMP D
C: i1 ← 40
   i2 ← 20
D: ··· ← i1 + i2
```

도면3



도면4

(가)

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

(나)

For all $s \in S$, $x \in S^*$,
$hd_0 x = \epsilon$, $hd_d x = x$ if $\|x\| \le d$
$hd_1 F = N$, $hd_d U_w = U_d$ if $d \le w$

$$hd_d(sx) = \begin{cases} hd_d s, & \text{if } d \le \|s\| \\ s(hd_{d-\|s\|}x), & \text{otherwise} \end{cases}$$

(다)

For all $s \in S$ and $\sigma \in D$,
$tl_d \bot = \bot$, $tl_d \top = \top$, $tl_0 \sigma = \sigma$
$tl_1 F = N$, $tl_d U_w = U_{w-d}$, if $d < w$

$$tl_d(s\sigma) \stackrel{\cdot}{=} \begin{cases} (tl_d s)\sigma, & \text{if } d < \|s\| \\ tl_{d-\|s\|}\sigma, & \text{otherwise} \end{cases}$$

(라)

For all $x, y \in S^*$, $x \cdot \bot = \bot$, $x \cdot y\top = hd_k(xy)\top$

(마)

$\bot \sqcup \sigma = \sigma$, $\top \sqcup \sigma = \top$
$s \sqcup t = t$ if $s \sqsubseteq t$ where $U_1 \sqsubseteq B \sqsubseteq I \sqsubseteq N$ and $U_2 \sqsubseteq F$
$s\sigma \sqcup t\tau = (s \sqcup hd_{\|s\|}t) \cdot (\sigma \sqcup tl_{\|s\|}(t\tau))$ if $\|s\| \le \|t\|$

도면5

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

$\mathcal{F}_e(s) = \sigma_2 \cdot tl_{\|\sigma_1\|}s$ where $type(e) = \sigma_1 \to \sigma_2$
$\mathcal{B}_e(s) = \sigma_1 \cdot tl_{\|\sigma_2\|}s$ where $type(e) = \sigma_1 \to \sigma_2$

도면6

입력코드	초기값	순방향 분석($\mathcal{F}_e$)	역방향 분석($\mathcal{B}_e$)	레지스터	가상스택	생성코드
S:	⊤	⊤	⊤		[]	
PUSH 1	⊥	$I\top$	$I\top$		$[1]$	
PUSH n	⊥	$II\top$	$II\top$		$[n,1]$	$(i_1 \leftarrow 1; i_0 \leftarrow n)$
A:	⊥	$II\top$	$II\top$	i_0, i_1	$[i_0, i_1]$	A:
$\texttt{DUP}_i$	⊥	$III\top$	$III\top$		$[i_4, i_4, i_1]$	$i_4 \leftarrow i_0$
PUSH 1	⊥	$IIII\top$	$IIII\top$		$[1, i_4, i_4, i_1]$	
CMP ≤	⊥	$BII\top$	$BII\top$		$[i_4 \le 1, i_4, i_1]$	
BCC C	⊥	$II\top$	$II\top$	i_2, i_3	$[i_4, i_1]$	$(i_3 \leftarrow i_1; i_2 \leftarrow i_4;)$ BCC $i_4 \le 1$ C
B:	⊥	$II\top$	$II\top$	i_2, i_3	$[i_2, i_3]$	B:
$\texttt{DUP}_i$	⊥	$III\top$	$III\top$		$[i_5, i_5, i_3]$	$i_5 \leftarrow i_2$
PUSH 1	⊥	$IIII\top$	$IIII\top$		$[1, i_5, i_5, i_3]$	
$\texttt{SUB}_i$	⊥	$III\top$	$III\top$		$[i_5 - 1, i_5, i_3]$	
PUSH t	⊥	$IIII\top$	$IIII\top$		$[t, i_5 - 1, i_5, i_3]$	
STI 1	⊥	$II\top$	$II\top$		$[i_5, i_3]$	$\texttt{M}[t] \leftarrow i_5 - 1$
$\texttt{MUL}_i$	⊥	$I\top$	$I\top$		$[i_3 * i_5]$	
PUSH t	⊥	$II\top$	$II\top$		$[t, i_3 * i_5]$	
LOI 1	⊥	$UI\top$	$II\top$		$[\texttt{M}_i[t], i_3 * i_5]$	
JMP A	⊥	$UI\top$	$II\top$	i_0, i_1	$[\texttt{M}_i[t], i_3 * i_5]$	$(i_1 \leftarrow i_5 * i_3; i_0 \leftarrow \texttt{M}_i[t];)$ JMP A
C:	⊥	$II\top$	$II\top$	i_2, i_3	$[i_2, i_3]$	
$\texttt{POP}_i$	⊥	$I\top$	$I\top$		$[i_3]$	