포인터와 자료를 중복하여 공간 효율을 높인 G-machine
(A Space-Efficient GM with Overlapping Pointers and Data)

박 홍 영 † 한 태 숙 ††
(HongYoung Park) (TaiSook Han)

요 약 G-Machine은 주어진 표현식을 그래프 축약을 통해서 계산한다. 이러한 축약 과정은 표현식을 그래프로 표현하기 위해 많은 그래프 공간을 필요로 한다. 본 논문에서는 그래프 공간을 효율적으로 사용할 수 있는 방법을 제안한다. 본 논문에서는 우회 노드 장소에 자료를 중복 사용하는 방법을 제안하여 G-Machine에서 필요로 하는 그래프 공간을 절약하며, 이를 위한 새로운 추상기계인 poGM(Pointer Overlapped GM)을 제안한다. poGM에서는 필요한 곳에만 상대 주소를 사용하여, 그래프 내부 노드와 단말 노드 모두에서 그래프 공간을 절약할 수 있다. 기억 장소 재활용 체계를 포함하는 시뮬레이션의 결과 벤치마크 프로그램에서 GM보다 총힙 사용량은 평균 32%, 최소힙은 평균 47.6% 적게 사용하였으며, 수행시간은 평균 30% 빠르게 수행되었다.

Abstract G-Machine requires much heap space during evaluation of given expressions with graph reduction. In order to reduce heap space, we propose a new space efficient abstract G-Machine called poGM(Pointer Overlapped GM) which uses a pointer-overlapping strategy. In general, poGM uses less relative address so that graph nodes are not splitted. Experimental results show that poGM reduces the average total heap space to 32%, the average minimal heap space to 47.6% and reduces the average runtime to 30% compared to GM.

1. 서 론

프로그램의 명확한 의미, 높은 차원의 추상화 지원으로 인한 빠른 프로그래밍으로 함수형 언어(functional language)는 명령형 언어(imperative language)와 다른 형태의 프로그램 작성법을 지원한다는 면에서 지속적인 주목을 받아 왔다[1]. 특히, 필요에 따른 계산법을 통해 프로그램 수행 순서까지 추상화한 지연 함수형 언어(lazy functional language)는 또 다른 측면의 프로그램 작성 방법을 지원한다[2]. 이를 구현하기 위해 람다 계산법(lambda calculus)에 기반을 두고 있는 함수형 언어를 효과적으로 수행할 수 있는 다양한 추상기계(abstract machine)들이 제안되었고, 이들 대부분은 그래프 축약(graph reduction) 방법에 기반을 두고 있다. 특히, 1984년 Johnsson 과 Augustsson에 의해 제안되어, 컴파일 방식을 통해 빠른 그래프 축약을 지원하는 GM[3,4]은 현재까지도 지연 함수형 언어 구현 방법의 하나로 사용되고 있다.

지연 함수형 언어에서 계산되지 않은 값은 그래프로 자연스럽게 나타낼 수 있다. 일반적으로 계산되지 않은 표현식을 나타내는 그래프는 그 표현식의 결과를 나타내는 그래프보다 커다란 공간에 저장되므로, 이를 저장하기 위해 그래프 축약 기계는 많은 공간을 필요로 한다. 이러한 그래프 공간을 줄이기 위해 ZGM[5]은 특별한 부호화와 태그 옮김(tag forwarding)을 통해 그래프의 저장 형태를 줄여 보다 작은 그래프 공간에 저장할 수 있게 하였다. 그러나, ZGM에서는 모든 노드에서 태그 옮김으로 인해 태그와 자료가 분리되며, 노드 공유시 태그 옮김 기법이 사용될 수 없다. 또한, ZGM에서 사용하는 기억 장소 재활용 체계는 깊이 우선 방식으로 재귀 호출에 따른 수행 시간의 부담이 따른다. 최근에

· 본 논문은 첨단정보기술센터를 통하여 과학재단의 지원을 받았음.
† 학생회원 : 한국과학기술원 전산학과
kypark@satrec.kaist.ac.kr
†† 종신회원 : 한국과학기술원 전산학과 교수
han@cs.kaist.ac.kr
논문접수 : 2001년 2월 19일
심사완료 : 2001년 6월 20일

발표된 논문[6]에서는 재귀 호출이 없는 너비 우선 방식을 사용하지만, 이것은 노드 복사 시에 자료와 태그의 위치를 바꾸어 저장하고, 검사 시에 다시 원상 복귀 시켜야 하는 실행 시간 부담이 존재한다. 본 논문에서는 이러한 문제가 없는 포인터와 자료를 중복하는 방법(pointer overlapping)을 사용하여, 태그와 자료의 태그 분리를 막고, 실행 시 공유된 노드에도 그래프 공간을 줄일 수 있는 그래프 축약 기계인 poGM을 제안한다. 그리고 실험을 통하여 그래프 공간이 줄어들고, 실행 시간이 줄어드는 것을 보인다.

본 논문의 구성은 다음과 같다. 제 2장에서는 본 논문과 관련된 GM, ZGM에 대하여 간단히 살펴보고, 제 3장에서는 본 논문에서 제안하는 poGM에 대하여 살펴본다. 그리고 제 4장에서는 실험을 통하여 GM, ZGM, poGM의 수행시 필요한 공간과 시간에 관하여 살펴보고, 제 5장에서 결론을 맺는다.

2. 관련 연구

각 추상기계들은 주어진 입력 언어를 수행하기 위하여, 입력 언어를 자신이 수행 가능한 언어로 변환 하며, 이를 수행하여 원하는 결과를 얻을 수 있다. poGM(pointer overlapped GM)의 비교 대상으로 다루는 GM과 ZGM도 역시 이와 같은 변환 과정을 거쳐 프로그램을 되는데, 이 장에서는 이들 추상기계에 대해서 알아본다.

2.1 GM

스웨덴 Charmers 공대에서 Augustsson과 Johnsson에 의해 고안된 GM[3,4]은 그래프 축약을 이용하여 프로그램을 수행하는 추상기계이다. GM의 입력으로 주어지는 프로그램은 자유 변수가 없는 함수인 조합자(combinator)로 이루어져 있다. 주어진 프로그램의 각 완전 조합자(supercombinator)는 G-code라는 GM의 중간 표현으로 변환되고, 이 G-code를 수행함으로써 그래프 축약을 수행한다. GM은 컴파일 규칙과 상태 변환 규칙으로 구성된다. 컴파일 규칙은 주어진 완전 조합자 프로그램을 G-code로 컴파일하기 위한 규칙이고, 상태 변환 규칙은 상태 변환을 통해 G-code를 수행하기 위한 규칙이다.

GM의 그래프 축약에 의한 수행 과정은 각 완전 조합자에 대하여 몸체에 해당하는 그래프를 생성하는 과정과 이를 축약하는 과정을 통해 그래프 축약이 이루어진다. 완전 조합자의 몸체 표현식에서 참조하는 변수는 스택에서 찾을 수 있는데, 이를 이용하여 몸체 표현식에 해당하는 그래프를 생성한 후, 결과로 만들어진 그래프에서 축약 가능한 표현식(redex: reducible expression)을 찾음으로써 그래프 축약이 계속된다. 이 과정을 나열하면 ①축약 가능 표현식 탐색, ② 스택에 인수 넣기, ③ 그래프 생성, ④축약 가능 표현식 갱신, ⑤스택에서 인수 빼내기로 나타낼 수 있으며, 이러한 과정은 더 이상 축약 가능한 표현식이 없을 때까지 반복 수행된다.

2.2 ZGM

ZGM은 생성되는 그래프 저장 형태의 크기를 줄여 힙 공간을 절약할 수 있도록 태그 옮김 방법으로 GM을 개선한 추상기계[5]이다. 힙 공간에 저장되는 그래프는 컴파일 규칙에 의해 생성된 G-code를 수행함으로써 생성하며, 이는 그래프를 압축하는 과정이 컴파일 시간에 이루어지게 하여 그래프 압축에 따른 실행 시간 부담을 없게 한다. 하지만, ZGM은 실행 시간에 압축된 그래프를 해석하는 부담이 발생한다. 또한, 태그 옮김 방법을 이용하므로, 각 노드는 태그와 자료를 구분하여 가지고 있게 되는 단점을 갖는다. 그리고, 노드 공유가 발생하였을 때, 태그 옮김 방법을 이용하는데 제한적이다.

GM에서 힙에 저장된 그래프 노드들은 태그와 한 개 이상의 자료로 이루어지며, 그래프를 이루기 위해 포인터의 연결된 구조로 이루어진다. 포인터는 표현식의 계산 결과를 공유하기도 하고 순환 자료 구조를 나타내기 위해서 사용되기도 하는데, 이들은 실행 시간에 연결된다. ZGM은 일시에 생성되는 그래프의 일부를 태그와 상대주소를 이용하여 한 워드에 저장하고, 트리 모양의 그래프에 대해서 순차 블록으로 그래프를 생성하여 그래프 크기를 줄인다.

ZGM은 태그를 옮겨 힙 공간을 절약하지만, 실행 시간에 힙 워드를 구분해야 하는 부담을 갖게 된다. 실행 시간에 어떤 힙 워드를 참조할 때, 어떤 태그를 갖고 있는지를 구분해야 한다. 이렇게 구분하는 일은 GM에서도 발생한다. 하지만, ZGM에서는 태그와 상대 주소가 합쳐진 워드에서 태그만을 구분하는 부담이 생긴다. 또한 노드의 자료 영역을 찾는 과정에서 ZGM은 상대 주소를 통해 자료 영역을 찾는 것이 필요한데, 이러한 실행상의 비용 부담이 비현실적이라고 생각될 수도 있지만, 실제 실행 결과에서는 수행 시간의 부담이 그렇게 크지는 않았다. 이는 단순히 그래프 크기가 작아지고, 힙 공간의 접근 횟수가 줄었음에 기인하는 것이다[5].

3. poGM

poGM은 ZGM과 마찬가지로 GM에서 생성되는 그래프 저장 형태의 크기를 줄여 힙 공간을 절약할 수 있도록 개선된 추상기계이다. poGM은 ZGM에서 그래프 노

드에 부여되는 태그에 부가적인 정보로 상대 주소만을 사용한 것을 확장하며, ZGM에서 발생하는 태그 옮김에 따른 태그와 데이타 공간의 분리를 막아서 컴파일 과정의 수행 시간을 줄이고, 컴파일 결과로 나온 코드의 실행 공간과 시간을 줄인다. 하지만, ZGM과 마찬가지로 실행 시간에 압축된 그래프를 해석하는 부담이 따른다. 그러나, 이러한 해석에 따른 비용도 ZGM이 가지고 있는 상대 주소보다 적은 수의 상대 주소를 갖기 때문에 해석 부담이 ZGM보다도 적다.

3.1 입력 언어 정의

p	::=	d^{*}
d	::=	$f\ x^{*} = e$
e	::=	$f \mid x \mid i \mid e_0\ e_1$
f, x	∈	이름
i	∈	정수 값

그림 1 입력 언어 정의

본 논문에서 사용하는 추상기계의 입력 언어를 정의한다. 입력 언어 구문의 정의는 [그림 1]에 정의되어 있다. 입력 프로그램 p는 하나 이상의 완전 조합자 d로 구성된다. 완전 조합자 d는 이름을 f로 하여 0개 이상의 인수를 갖고 표현식 e를 반환하는 형태로 정의된다. 본 논문에서 프로그램 전체를 통하여 같은 이름을 갖는 완전 조합자는 없다고 가정한다. 표현식 e는 완전 조합자 이름 f , 완전 조합자의 인수(또는 변수) x, 정수 값 i , 표현식 e_0 에 e_1이 적용된 형태로 정의된다.

이름으로 사용되는 것으로 완전 조합자 이름 f, 변수 이름 x가 있다. 정수 값 표현은 i로 대표된다. 정수 값과 이름은 서로 구분되어야 하므로, 이름과 정수 값 구문에 해당하는 집합은 서로소(disjoint)라고 가정한다.

3.1.1 그래프 표현 형식

GM의 모든 노드의 표현은 태그와 자료의 형태를 취하고 있으며, 자료의 위치는 절대 주소나 우회 노드로서 표현된다. 이러한 표현에서 적용 노드는 자료의 위치를 나타내기 위하여 절대 주소를 갖고 있으며, 이로 인해 표현식을 표현하기 위한 전체 그래프 표현은 많은 적용 노드를 필요로 하고, 이는 많은 수의 절대 주소를 위한 많은 힙 장소를 필요로 함을 의미한다. ZGM에서는 태그 장소에 자료의 상대 주소 정보를 갖는다. 이것은 태그가 자료의 위치를 나타내는 역할을 하는 것으로 해석할 수 있는데, 대부분의 태그가 상대 주소 정보를 가지므로 각 노드는 자료를 찾기 위해 태그의 상대 주소를 참조하여야 하는 부담을 갖게 된다. poGM도 태그를 이용한다는 면에서는 ZGM과 같다고 할 수 있다. 그러나, 일반적인 노드는 자료의 위치를 찾기 위해 GM과 유사하게 상대 주소를 참조하는 것이 필요하지 않다.

GM과 ZGM, poGM의 그래프 표현 형식을 나타내기 위해 적용 노드의 그래프 표현 형식을 살펴 보겠다. 각 추상기계에 따라서 적용 노드의 표현 형식이 달리 표현된다. GM에서 적용 노드는 태그와 왼쪽 노드를 나타내는 워드와 오른쪽 노드를 나타내는 워드를 갖는 형태로 표현된다. ZGM에서 적용 노드는 태그 부분에 상대 주소로 태그의 왼쪽 노드를 나타내는 워드를 갖고 왼쪽 노드 바로 다음 워드에 오른쪽 노드를 갖는 형태로 표현 된다. poGM에서 적용 노드는 태그 부분에 상대 주소로 오른쪽 노드를 나타내는 워드를 갖고 왼쪽 노드는 바로 이 태그 워드 다음에 존재하게 된다.

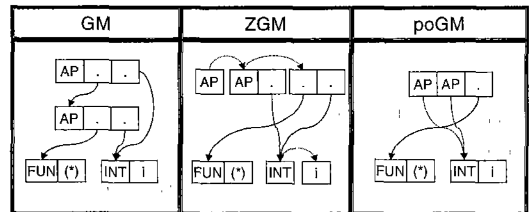


그림 2 각 추상기계에 따른 그래프 표현

[그림 2]는 한 개의 정수 값 i를 제곱하는 표현식 i * i 를 그래프 노드를 표현한 것이다. GM에서는 적용(AP) 노드의 왼쪽 노드와 오른쪽 노드가 모두 절대 주소를 나타내는 표현이며, 공유된 정수값 i는 절대 주소로 공유한다. ZGM에서는 적용 노드의 태그에 왼쪽 노드를 나타내는 상대 주소가 있고 왼쪽 노드 바로 뒤에 오른쪽 노드가 있게 된다. 이때 적용 노드의 오른쪽 노드는 정수값 i 가 공유되는 상황이므로, 절대 값으로 참조하게 된다. poGM에서는 적용 노드의 태그에 오른쪽 노드를 나타내는 상대주소가 있으므로, 공유하는 정수값 i를 상대주소로 가르키면 된다.

[그림 2]에서 그래프 공간을 사용하는 면에서 보면 GM 10개, ZGM 9개, poGM 7개의 워드를 사용하는 것을 볼 수 있으며, 이는 poGM이 공유시 공간 효율이 좋다는 것을 의미한다. 이때 ZGM이 많은 공간을 차지하는 이유는 ZGM이 공유시 태그 옮김을 할 수 없기 때문이다. 또한 자료를 참조하는 면을 살펴보면, GM은 실선으로 표시되는 절대 주소를 4개 가지고 있으며, ZGM은 절대 주소 3개, 점선으로 표시되는 상대주소 3개를 가지고 있는 반면, poGM의 경우 절대 주소 1개, 상대 주소 2개만을 사용하는 것을 알 수 있다. 이것은 자료를

다루기 위해 노드의 값을 참조하는 경우에 poGM이 GM이나 ZGM보다 시간 부담이 적다는 것을 의미한다.

3.2 명령어와 그래프 노드

(a) 명령어		(b) 노드 종류
ALLOCSEQ	*c*	
COPY	*w*	
DONE		(AP , *o*)
POP	*k*	(INT)
PUSH	*k*	(IND , *n*)
PUSHFUN	*f*	(*integer*)
UNWIND		(FUN, *arity*) *code*
UPDATE	*k*	

그림 3 poGM의 명령어와 그래프 노드 종류

poGM의 명령어는 ZGM과 호환을 이루는 형태로 [그림 3]에 정의되었다. 각각의 명령어의 수행 의미는 상태 변환 규칙에 나타나 있다. poGM의 그래프 노드 종류는 ZGM과 마찬가지로 개별적인 워드 단위로 분리되지만, 대부분의 노드의 구조는 해체되지 않는다. (AP,*o*)는 태그와 상대주소의 순서쌍을 나타낸다는 면에서 ZGM과 구조는 같지만 poGM에서 상대주소 *o*는 오른쪽 노드의 상대 주소를 나타내는 것이 다르다. (INT)노드는 태그와 자료의 분리가 없으므로 상대주소가 없이 표현하였으며, ZGM과 유사하게 표현하면, (INT, 1)노드로 표현 가능하다. (FUN, *arity*)노드는 태그의 분리가 일어나지 않기 때문에 *arity*같이 한 워드를 모두 사용하지 않는 자료에 대해서 축약이 가능하다는 것을 나타내기 위해서 표시하였다.

3.3 poGM의 코드 생성

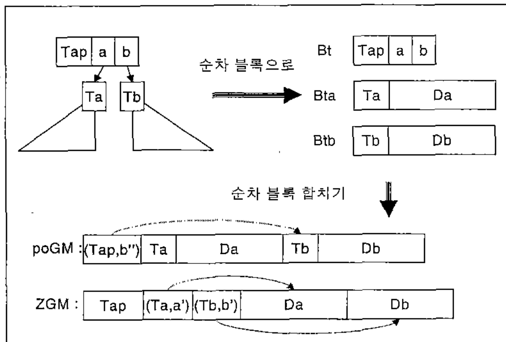


그림 4 순차 블록을 이용하여 그래프를 압축하는 과정

그림[4]는 트리를 순차 블록의 형태로 변환하는 과정을 나타낸다. 첫 번째 단계는 트리의 일부분을 순차 블록 형태의 그래프로 변환하는 과정을 나타내며, 각 블록 Bt, Bta와 Btb는 트리 Tap, Ta, Tb의 순차 블록 표현이다. 두 번째 단계는 나누어진 순차 블럭을 압축하면서 합치는 과정이다. ZGM은 이를 위하여 태그 옮김의 방법을 사용하는데, Bta에 있는 태그 Ta가 a 영역으로 옮겨지고, Tb가 b 영역으로 옮겨지면서 a가 (Ta,a′) 형태로, b가 (Tb,b′) 형태로 압축된다. poGM은 데이타 자체가 옮겨지는 것으로 (Tap,b′′)형태로 옮겨진다. 이렇게 하면, ZGM에서는 두 개의 상대 주소가 생기는 반면, poGM에서는 한 개의 상대 주소가 생기므로 자료를 참조하는 경우에 상대 주소를 이용하는 횟수가 줄어 자료를 찾는데 있어서 ZGM보다 시간적인 성능 향상이 가능하게 한다. 또한 poGM에서는 태그를 옮기는 절차가 필요하지 않기 때문에 컴파일러에서 코드 생성 시 성능 향상을 기대할 수 있다. 게다가 poGM에서는 단말 노드 자체의 표현에서 태그를 저장하고 남는 공간에 여분의 자료를 저장하는 것이 가능하기 때문에, ZGM보다 그래프를 저장하기 위한 공간적인 측면에서도 성능 향상을 기대할 수 있다. 물론 FUN 노드에 대해서 ZGM은 태그 포워딩 하지 않으므로, FUN 노드에 대해서는 poGM과 같이 저장 장소를 줄이는 것이 가능하다.

3.4 컴파일 규칙

$$\mathbf{F}\llbracket f\ x_1\ \dots\ x_k = e \rrbracket = \mathbf{R}\llbracket e \rrbracket\,[\, x_1=0,\ \dots\ x_k=(k-1)\,]\ k$$

$$\mathbf{R}\llbracket e \rrbracket\ \rho\ k = \mathbf{C}\llbracket e \rrbracket\ \rho \ {+}{+}\ [\ \text{UPDATE}\ k,\ \text{POP}\ k,\ \text{UNWIND}\ ]$$

$$\mathbf{C}\llbracket e \rrbracket\ \rho = [\ \text{ALLOCSEQ}\ (\ \mathbf{N}\llbracket e \rrbracket\ \rho\ )\ ]$$

$$\mathbf{N}\llbracket f \rrbracket\ \rho = [\ \text{PUSHFUN}\ f\ ]$$

$$\mathbf{N}\llbracket x \rrbracket\ \rho = [\ \text{PUSH}\ (\ \rho\ x\ )\ ]$$

$$\mathbf{N}\llbracket i \rrbracket\ \rho = [\ \text{COPY}\ (\ \text{INT}\ )\ ,\ \text{COPY}\ i\ ]$$

$$\mathbf{N}\llbracket e_0\ e_1 \rrbracket\ \rho = [\ \text{COPY}\ (\text{AP},o)\ ]\ {+}{+}\ \mathbf{N}\llbracket e_0 \rrbracket\ \rho\ {+}{+}\ \mathbf{N}\llbracket e_1 \rrbracket\ \rho \quad \textit{where} \quad o = |\mathbf{N}\llbracket e_0 \rrbracket\ \rho| + 1$$

그림 5 poGM의 컴파일 규칙

주어진 입력 언어를 G-code로 변환하기 위한 poGM의 컴파일 규칙이 [그림 5]에 정의되어 있다. poGM의 컴파일 규칙은 GM과 ZGM에서 사용하는 컴파일 규칙과 유사하게 **F**, **R**, **C**, **N**규칙으로 구성되어 있다.

F는 완전 조합자 하나를 컴파일하기 위한 규칙이며, **R**

은 완전 조합자의 반환값을 정의하기 위한 규칙으로 GM, ZGM과 같다. **C**는 그래프를 연속적으로 생성하기 위한 규칙으로 ZGM과 유사하게 정의되었지만, 상대 주소를 필요로 하지 않는다. **N** 규칙은 각 노드에 따른 그래프를 생성하기 위한 규칙으로 ZGM과 유사하지만, **C**로부터 표현식 e와 변수 환경 ρ만을 전달받는 것이 ZGM과 다르며, 이는 GM의 **C** 규칙과 유사하다고 할 수 있다.

본 논문에서는 정수에 관한 그래프를 생성하기 위하여 COPY (INT), COPY (i) 라는 명령어 형태를 사용하지만, 의미상으로는 힙에 두 워드를 설정한다는 면에서 같다. 적용 표현식 e_0 e_1인 경우, 해당 태그와 상대 주소 정보 o를 인수로 갖는 COPY 명령어를 생성하고, 표현식 e_0 에 대한 G-code를 재귀적으로 생성하고, 표현식 e_1 에 대한 G-code를 재귀적으로 생성한 명령어 리스트를 접합하여 AP 노드를 나타낸다. 이때 필요로 하는 상대 주소는 오른쪽 노드가 있을 상대 주소를 나타내며, 리스트 길이를 알기 위한 함수(length) 호출이 컴파일 과정에서 필요하다.각 추상기계별로 주된 차이가 나는 부분을 [표 1]에 표시하였다. 적용 노드를 생성하는 규칙을 다른 추상기계와 비교하면, GM은 e_1에 대한 G-code를 생성 한 후에, e_0에 대한 G-code를 생성하고, MKAP 라는 명령어로 AP 노드를 생성한다. 이때 환경 함수에 있는 변수들의 위치를 변경해 주어야 하는 부분인 $\rho+1$ 이 필요하다. ZGM은 e_0, e_1에 대한 G-code를 생성한 후, 태그 옮김에 따라 명령어를 재배열을 하게 된다. 또한 e_1의 명령어 생성시에 e_0의 그래프 크기에 대한 상대 주소를 계산하여야 하는 부담을 갖게 된다. 반면, poGM에서는 길이를 알기 위해 함수(length)를 사용하는 경우 ZGM에서와 같이 상대 주소를 계산하여 전달해 주어야 할 필요가 없으며, 재귀 호출 과정에서 생성한

표 1 컴파일 규칙 비교

poGM	$\mathbf{C}[\![e]\!]\rho$	$= [\ \text{ALLOCSEQ}\ (\ \mathbf{N}[\![e]\!]\rho\ )\ ]$
	$\mathbf{N}[\![i]\!]\rho$	$= [\ \text{COPY}\ (\ \text{INT}\ ),\ \text{COPY}\ i\ ]$
	$\mathbf{N}[\![e_0\ e_1]\!]\rho$	$= [\ \text{COPY}\ (\text{AP},o)\ ] \mathbin{++} \mathbf{N}[\![e_0]\!]\rho \mathbin{++} \mathbf{N}[\![e_1]\!]\rho$
		$\textit{where}\quad o = \lvert \mathbf{N}[\![e_0]\!]\rho \rvert + 1$
GM	$\mathbf{C}[\![i]\!]\rho$	$= [\ \text{PUSHINT}\ i]$
	$\mathbf{C}[\![e_0\ e_1]\!]\rho$	$= \mathbf{C}[\![e_1]\!]\rho \mathbin{++} \mathbf{C}[\![e_0]\!](\ \rho+1\ ) \mathbin{++} [\text{MKAP}]$
ZGM	$\mathbf{C}[\![e]\!]\rho$	$= [\ \text{ALLOCSEQ}\ (\ \mathbf{N}[\![e]\!]\rho\ 1)\ ]$
	$\mathbf{N}[\![i]\!]\rho\ o$	$= [\ \text{COPY}\ (\ \text{INT},\ o\ ),\ \text{COPY}\ i\ ]$
	$\mathbf{N}[\![e_0\ e_1]\!]\rho\ o$	$= [\ \text{COPY}\ (\ \text{AP}\ ,\ o\ ),\ hd_0\ ,\ hd_1\ ] \mathbin{++} tl_0 \mathbin{++} tl_1$
		$\textit{where}\ hd_0 : tl_0 = \mathbf{N}[\![e_0]\!]\rho\ 2$
		$hd_1 : tl_1 = \mathbf{N}[\![e_1]\!]\rho\ \lvert tl_0 \rvert + 1$

명령어 리스트를 태그 옮김을 위해 분할하고 재조합하는 과정이 필요 없다.

3.5 상태 변환 규칙

주어진 poGM 명령어를 수행하기 위한 방법으로 상태 변환 규칙을 [그림 6]과 같이 정의하였다. 이는 GM, ZGM과 유사한 형태로 이루어졌다. poGM의 상태는 명령어 리스트, 스택, 그래프 저장소, 완전 조합자 참조 리스트의 4가지 원소로 구성된다. poGM 수행을 위한 초기 상태는 < c_{init},[],G_{init},E_{init} > 로 주어진다. 초기 상태의 명령어 목록 c_{init} 은 [ALLOCSEQ [PUSHFUN *main*], UNWIND]이다. 초기 스택은 비어 있고, 초기 그래프 저장소 G_{init}에는 완전 조합자가 컴파일되어 FUN 노드 형태로 저장되고, 초기 환경 E_{init} 에는 완전 조합자 이름에 해당하는 FUN노드의 주소가 저장된다.

poGM의 상태 변환 규칙은 모두 12개의 규칙으로 이루어져 있다. 노드의 구조나 컴파일 규칙 등을 GM, ZGM과 유사하게 설계하였으므로, ALLOCSEQ, PUSHFUN, PUSH, COPY, DONE, UPDATE, POP 의 대부분 명령어들은 ZGM과 같다. 하지만 UNWIND에서 각 노드를 처리하는 방법이 다르다.

각 추상기계에 따른 주된 상태 변환 차이를 [표 2]에 나타내었다.

명령어가 UNWIND이고, 스택 최상위 원소 n이 가리키는 그래프가 (AP, o)인 경우, poGM은 적용할 함수를 나타내는 그래프 주소 (n+1)을 스택에 넣은 후, 계속 UNWIND를 수행한다. 이는 GM에서 그래프 내의 [n=AP n_0 n_1]에서 n_0을 찾는 과정과 비교되며, ZGM 에서 그래프 내의 [n= (AP o)]를 이용하여 자료를 찾는 과정과 비교된다. GM에서는 적용 노드의 주소 n에서 1을 더한 후에 그것이 가리키는 메모리를 참조하여 n_0값을 스택에 입력하며, ZGM에서는 적용 노드의 주소 n이 가지고 있는 값에서 상대 주소 o를 추출하여 n과 합한 값 n+o를 스택에 입력하는 작업이다. 하지만, poGM 에서는 단순히 n+1을 스택에 입력하기만 하므로 GM, ZGM에 비하여 실행 시간을 단축시킬 수 있다.

명령어가 UNWIND이고, 스택 최상위 원소 n이 가리키는 그래프가 (FUN , k) c' 인 경우, 함수의 인수 개수 k만큼의 노드가 스택에 있으면, 스택 상위에서 (k+1) 번째 적용 노드 n_k만을 남기고 나머지 노드에 대하여 이들 적용 노드의 인수 부분 a_i를 스택에 넣는다. 이때 적용 노드 n_k는 함수 적용 결과로 갱신될 노드를 가리킨다. 적용 노드의 오른쪽 노드 주소 a_i를 스택에 입력하는 경우, 스택을 따라 내려가면서 노드를 찾고, 그것이 가리키는 오른쪽 노드 주소를 스택 값으로 바꾸게

1.	$<$	ALLOCSEQ c' : c	,	s	,	G	,	E	$>$
⇒	$<$	c' : DONE : c	,	$0{:}n$: s	,	$G[n = alloc(\lvert c' \rvert)]$	,	E	$>$
2.	$<$	PUSHFUN f : c	,	$k{:}n$: s	,	G	,	$E[f=n_f]$	$>$
⇒	$<$	c	,	$(k+1){:}n$: s	,	$G[n+k=(\mathrm{IND}, n_f)]$	,	E	$>$
3.	$<$	PUSH i : c	,	$k{:}n{:}n_0{:}...{:}n_k$: s	,	G	,	E	$>$
⇒	$<$	c	,	$(k+1){:}n{:}n_0{:}...{:}n_k$: s	,	$G[n+k=(\mathrm{IND}, n_i)]$	,	E	$>$
4.	$<$	COPY w : c	,	$k{:}n$: s	,	G	,	E	$>$
⇒	$<$	c	,	$(k+1){:}n$: s	,	$G[n+k=w]$	,	E	$>$
5.	$<$	DONE : c	,	$k{:}n$: s	,	G	,	E	$>$
⇒	$<$	c	,	n : s	,	G	,	E	$>$
6.	$<$	UNWIND : c	,	n : s	,	$G[n=(\mathrm{IND}\ n')]$	,	E	$>$
⇒	$<$	c	,	n' : s	,	G	,	E	$>$
7.	$<$	UNWIND : c	,	n : s	,	$G[n=(\mathrm{AP}, o)]$	,	E	$>$
⇒	$<$	UNWIND : c	,	$n+1{:}n$: s	,	G	,	E	$>$
8.	$<$	UNWIND : c	,	n : s	,	$G[n=(\mathrm{INT})]$	,	E	$>$
⇒	$<$	c	,	n : s	,	G	,	E	$>$
9.	$<$	UNWIND : c	,	$n{:}n_1{:}...{:}n_k$: s	,	$G[n=(\mathrm{FUN}, k)\ c']$	,	E	$>$
⇒	$<$	c'	,	$a_1{:}...{:}a_k{:}n_k$: s	,	G	,	E	$>$
		where $G[n_1=(\mathrm{AP},o_1),...,n_k=(\mathrm{AP},o_k)]$ *and* $a_i = n_i+o_i$ $(i=1 ... k)$							
10.	$<$	UNWIND : c	,	$[n{:}n_1{:}...{:}n_{k'}]$	,	$G[n=(\mathrm{FUN}, k)\ c']$	,	E	$>$
⇒	$<$	c	,	$[n{:}n_1{:}...{:}n_{k'}]$	,	G	,	E	$>$
		where $k' < k$							
11.	$<$	UPDATE k : c	,	$n{:}n_0{:}...{:}n_k$: s	,	G	,	E	$>$
⇒	$<$	c	,	$n_0{:}...{:}n_k$: s	,	$G[n_k=(\mathrm{IND}, n)]$	,	E	$>$
12.	$<$	POP k : c	,	$n_1{:}...{:}n_k$: s	,	G	,	E	$>$
⇒	$<$	c	,	s	,	G	,	E	$>$

그림 6 poGM 의 상태 변환 규칙

표 2 상태 변환 규칙 비교

poGM		$<$	UNWIND : c	, n : s	, $G[n=(\mathrm{AP}, o)]$	,	E	$>$	
	⇒	$<$	UNWIND : c	, $n+1{:}n$: s	, G	,	E	$>$	
		$<$	UNWIND : c	, $n{:}n_1{:}...{:}n_k$: s	, $G[n=(\mathrm{FUN}, k)\ c']$	,	E	$>$	
	⇒	$<$	c'	, $a_1{:}...{:}a_k{:}n_k$: s	, G	,	E	$>$	
			where $G[n_1=(\mathrm{AP},o_1),...,n_k=(\mathrm{AP},o_k)]$ *and* $a_i = n_i+o_i$ $(i=1 ... k)$						
GM		$<$	UNWIND : c	, n : s	, $G[n=\mathrm{AP}\ n_0\ n_1]$	,	E	$>$	
	⇒	$<$	UNWIND : c	, $n_0{:}n$: s	, G	,	E	$>$	
		$<$	UNWIND : c	, $n{:}n_1{:}...{:}n_k$: s	, $G[n=\mathrm{FUN}\ k\ c']$	,	E	$>$	
	⇒	$<$	c'	, $a_1{:}...{:}a_k{:}n_k$: s	, G	,	E	$>$	
			where $G[n_i=\mathrm{AP}\ _\ a_i, ..., n_k=\mathrm{AP}\ _\ a_k]$ $(i = 1 ... k)$						
ZGM		$<$	UNWIND : c	, n : s	, $G[n=(\mathrm{AP}, o)]$	,	E	$>$	
	⇒	$<$	UNWIND : c	, $n+o{:}n$: s	, G	,	E	$>$	
		$<$	UNWIND : c	, $n{:}n_1{:}...{:}n_k$: s	, $G[n=\mathrm{FUN}\ k\ c']$	,	E	$>$	
	⇒	$<$	c'	, $a_1{:}...{:}a_k{:}n_k$: s	, G	,	E	$>$	
			where $G[n_1=(\mathrm{AP},o_1),...,n_k=(\mathrm{AP},o_k)]$ *and* $a_i = n_i+o_i+1$ $(i=1 ... k)$						

되는데, 각 기계마다 스택을 따라 가는 것과 스택 값을 바꾸는 것은 비용이 같다. 하지만, 각 기계에 따라서 오른쪽 노드 주소를 찾는 것이 다른다. GM에서는 n_i=AP _ a_i 로 표시되므로 오른쪽 노드 a_i로 접근하기 위해 n_i+2 를 수행하고 그것이 참조하는 메모리 값인 a_i 를 찾는다. ZGM에서는 n_i = (AP, o) 로 표현되므로 상대 주소를 이용하여 왼쪽 노드의 주소 (n_i+o)를 알아내고, 거기에 오른쪽 노드가 있는 주소(n_i+o+1)를 찾는다.

poGM에서는 n_i = (AP, o)로 ZGM과 같이 표현되지만, 상대 주소를 이용하여 오른쪽 노드의 주소(n_i+o)를 찾는다. 그러므로, 오른쪽 노드를 찾아서 함수에 적용될 인수를 구성하는 면에서도 ZGM보다 약간의 실행 시간을 단축시킬 수 있다.

4. 실험 및 분석

poGM의 성능을 평가하기 위하여 GM, ZGM과 실험을 통하여 비교하였다. 본 논문에서는 각 추상기계의 명령어를 C언어로 번역하고, 생성된 C프로그램을 C로 작성한 실행 시간 시스템과 함께 컴파일하여 수행 코드를 생성한다. C언어로의 번역은 기존 연구 방법[5][9][10]을 참조하여 구현하였다. 가능하면 정확한 비교를 위하여 GM, ZGM, poGM은 서로 많은 부분에서 유사한 구조로 설계하여, 가능한 많은 부분을 공유하도록 하였다. 시뮬레이션에서는 PENTIUMII-466, 메모리 128Mbyte를 장착한 PC를 이용하였으며, OS로는 LINUX의 일종인 miziOS 1.0을 사용하였다. C 컴파일러로는 GNU C 컴파일러 egcs-2.91.66을 사용하였으며, -O2 옵션으로 컴파일 하였다. 벤치마크 프로그램으로는 Haskell 벤치마크인 nofib 벤치마크[11] 프로그램 중 소규모 프로그램(imaginary subset) 몇 개를 이용하였다.

평가 기준으로는 컴파일된 프로그램의 공간 효율, 시간 효율로 하였다. 공간 효율 측정에서 고려해야 할 사항으로 본 논문에서는 실행시 사용하는 총 힙 사용량, 수행할 수 있는 최소 힙 사용량을 측정하였다. 시간 효율 측정으로는 외부 환경을 같게 하기 위해서 힙의 크기가 일정할 경우 각 추상기계의 수행 속도를 측정하였다.

4.1 힙 사용량

[표 3]과 [표 4]에서 각 벤치마크 프로그램에 대한 3가지 추상기계에서의 총 힙 사용량과 최소힙 사용량을 비교하였다.

총 힙 사용량을 나타내는 [표 3]은 각 추상기계들이 컴파일된 프로그램을 수행하면서 할당한 힙의 크기를 표시하며, 이 이상의 힙이 주어진 경우 기억 장소 재활용 체계의 호출은 일어나지 않는다. 실험 결과에 따르면 poGM이 GM보다 평균 32% 정도 줄고, ZGM보다도 프로그램에 따라 2% 에서 9 % 정도 차이를 보이며 평균 4.7% 줄어들었다. 이것은 poGM의 그래프 노드 표현이 ZGM보다 그래프 공간을 절약할 수 있음을 의미한다. 특히, ZGM에 비하여 그래프 공간이 절약된 것은 그래프의 단말 노드 표현에 따른 공간 절약과 공유에 따른 공간 절약으로 볼 수 있다.

최소 힙 사용량을 나타내는 [표 4]는 프로그램을 수행하는데 필요한 최소 힙 워드의 크기를 나타내고 있다. 최소 힙의 작다는 것은 작은 공간에서 프로그램 수행이 가능하다는 것을 의미하며, 메모리 공간에 제약이 있는 시스템에서 더욱 효과적으로 사용 가능함을 의미한다. poGM은 GM보다 평균 47.6 %, ZGM보다도 22.6% 정도 최소 힙을 적게 사용함을 알 수 있다. 이것은 poGM에서 기억 장소 재활용 시에 공유 노드에 대한 효율적인 공간 활용을 가능하게 하므로 적은 공간에서 수행이 가능하다고 볼 수 있다.

총 힙 사용량과 최소 힙 사용량에 관한 실험을 통하여 poGM은 힙 공간을 사용하는 면에서 GM과 ZGM보다도 매우 공간 효율적이라는 것을 알 수 있다.

표 3 총 힙 사용량

(단위, word)

	exp	nfib	primes	queens	tak
GM	121,069,155	164,247,892	2,054,203	124,151,517	195,728,858
ZGM	88,793,471	115,781,380	1,584,609	94,087,377	135,888,242
poGM	80,727,079	113,088,733	1,475,011	91,002,419	133,394,825
(G-po)/G	33.3%	31.1%	28.2%	26.7%	31.8%
(Z-po)/Z	9.1%	2 3%	6 9%	3.3%	1.8%

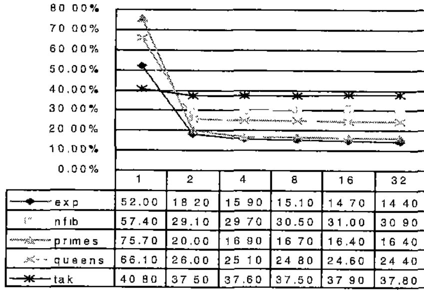
표 4 최소 힙 사용량

(단위, word)

	exp	nfib	primes	quuens	tak
GM	118,489	1,974	79,195	107,266	10,754
ZGM	79,059	1,453	45,898	71,443	7,911
poGM	52.795	1,348	38,807	50,944	5,654
(G-po)/G	55.44%	31.71%	51.00%	52.51%	47.42%
(Z-po)/Z	33.22%	7.23%	15.45%	28.69%	28.53%

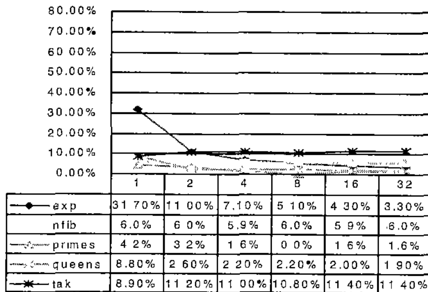
4.2 동일한 힙 공간이 주어진 경우

힙 공간이 동일한 상태에서 각 프로그램, 각 추상기계의 수행 시간 효율을 실험을 통하여 비교하였다. 여기서 첫번째 행의 숫자 1은 한 단위를 나타내는데, 실험에 사용된 입력 프로그램의 3개의 추상기계에서 사용하는 최소 힙 중에서 가장 큰 것으로 하였으며, 그 값은 각 프로그램의 GM의 최소 힙 크기이다. [그림 7]와 [그림 8]은 각 값들은 3번의 수행을 하고 평균을 낸 실행한 시간을 비율로 표시하였다.



	1	2	4	8	16	32
exp	52.00	18.20	15.90	15.10	14.70	14.40
nfib	57.40	29.10	29.70	30.50	31.00	30.90
primes	75.70	20.00	16.90	16.70	16.40	16.40
queens	66.10	26.00	25.10	24.80	24.60	24.40
tak	40.80	37.50	37.60	37.50	37.90	37.80

그림 7 (GM-poGM)/GM 수행속도



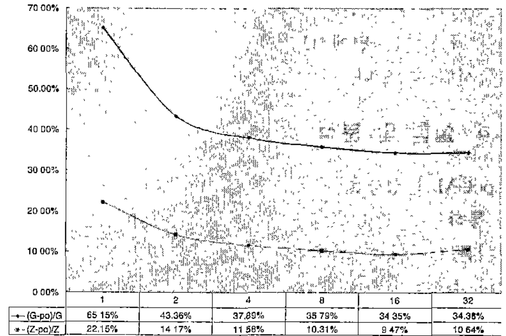
	1	2	4	8	16	32
exp	31.70%	11.00%	7.10%	5.10%	4.30%	3.30%
nfib	6.0%	6.0%	5.9%	6.0%	5.9%	6.0%
primes	4.2%	3.2%	1.6%	0.0%	1.6%	1.6%
queens	8.80%	2.60%	2.20%	2.20%	2.00%	1.90%
tak	8.90%	11.20%	11.00%	10.80%	11.40%	11.40%

그림 8 (ZGM-poGM)/ZGM 수행속도

[그림 7]은 벤치마크 프로그램에 대해 각 추상기계를 동일한 힙 크기 하에서 수행 시켰을 때의 GM에 대한 수행 속도 향상을 나타낸다. 힙 공간의 크기가 1 인 경우, 힙이 적은 공간에서 프로그램이 수행 되므로 힙 크기의 차이에 따른 기억 장소 재활용 체계의 호출 횟수의 차가 커지므로, poGM의 경우 GM보다 상대적으로 최고 75%정도 높은 속도 향상을 보인다. 그러나, 적당한 힙 공간이 주어졌을 때 각 프로그램에 대하여 GM보다 10% ~ 40 % 정도, 전체 평균 30%의 수행 시간이 향상되었다. 이것은 기억 장소 재활용 체계 보다는 순수 수행 시간에 좌우되기 때문인데, poGM이 GM보다 수행 속도가 빠른 것은 GM에서의 많은 메모리 참조 때문이다. [그림 8]은 ZGM에 대한 속도 향상을 보이고 있다. 힙 공간이 적은 경우 poGM은 ZGM에 비해 최고 31% 까지 속도 향상되었다. 그러나 힙 크기가 적당할 때 ZGM 보다는 1.6% ~ 11% 내외, 평균 6.5% 성능 향상을 보인다. poGM이 ZGM보다 빠른 이유는 수행 시간에 참조하여야 하는 상대 주소가 적기 때문이라고 생각된다.

본 실험 결과를 통해 poGM은 힙 공간이 같은 경우, 수행 시간 면에서도 GM이나 ZGM보다 효율적이라는 것을 알 수 있다.

4.3 힙 공간에 따른 기억 장소 재활용 체계의 호출 횟수



	1	2	4	8	16	32
(G-po)/G	65.15%	43.36%	37.89%	35.79%	34.35%	34.38%
(Z-po)/Z	22.15%	14.17%	11.58%	10.31%	9.47%	10.64%

그림 9 힙 크기에 따른 GC수 변화율

주어진 힙 공간에 대해 프로그램을 수행 함에 있어서 힙 공간을 분류하면 충분한 공간, 적당한 공간, 적은 공간, 부족한 공간으로 나눌 수 있다. 힙 공간이 충분하다는 것은 기억 장소 재활용 체계 호출 없이 프로그램이 수행 가능하다는 것을 의미하며, 힙 공간이 부족하다는 것은 기억 장소 재활용 체계를 호출하여도 프로그램이 수행 결과를 얻을 수 없는 것을 의미하며, 적다는 것은 프로그램 수행 결과를 얻기 위해서 결과를 얻기 위한 연산하는 시간보다 기억 장소 재활용 체계의 호출에 사용된 시간이 차지하는 비율이 많다는 것을 의미하며, 적당하다는 것은 기억 장소 재활용 체계의 호출에 사용된 시간보다 연산에 소요되는 시간이 차지하는 비율이 높다는 것을 의미한다.

만일, 같은 그래프를 축약을 하는데, 힙 공간을 적게 사용하는 추상 기계 Little 과 많이 사용하는 Big이 있다고 하자. 같은 힙, 같은 프로그램이 주어졌을 때, Big에서 수행하는 프로그램은 적은 힙 공간에 위치하더라도, Little에서 수행하는 프로그램은 적당한 공간에 있을 수 있다. 이것은 같은 힙이 주어졌을 때 Little에서 한 번 기억 장소 재활용 체계를 수행하는 시간이 Big보다 약간 크다고 하더라도, 주어진 프로그램을 수행하는 총 시간 면에서 Little이 이득이 될 수 있다는 것을 의미한다. 그러므로 힙 공간의 변화에 따른 기억 장소 재활용 체계의 호출 수와의 변화를 알아보는 것도 의미 있는 일이다.

[그림 9]은 프로그램 exp에 대하여 각 추상기계에 따른 주어진 힙 공간의 크기와 기억 장소 재활용 체계의 호출 수의 비율을 나타내고 있다. 위의 선은 GM에 대한 호출 수 감소 비율을, 아래 선은 ZGM에 대한 호출 수

감소 비율을 나타내고 있다. 여기서 감소 비율을 평균적으로 살펴보는 것 보다는 힙 공간 분류에 따라 살펴보겠다. 대략적으로 2 이하의 공간은 힙이 적은 공간으로, 그 이상의 공간은 적당한 경우로 볼 수 있다. 적은 힙 공간에 대해서 poGM은 GM보다 40% 이상, ZGM보다 14% 이상 호출수가 감소하였고, 적당한 공간에서 GM보다 35% 정도, ZGM보다 10% 정도 호출수가 감소하였다. 이것은 poGM이 기억 장소 재활용 체계를 수행하면서, 그래프에서 공유되는 부분까지도 찾아서 그래프 공간을 효율적으로 사용하기 때문이다. 기억 장소 재활용 체계와 함께 수행한 시간을 함께 살펴보면, GM의 최소 힙에서 기억 장소 재활용 체계의 수는 65%정도가 줄어든 반면, 수행 속도는 52%밖에 줄지 않았다. 힙 크기가 같은 경우, poGM은 GM보다 압축된 형태로 그래프를 가지게 되므로, 같은 공간에 더 많은 그래프 정보가 있게 되어 있다. 그러므로, 같은 힙 공간을 옮기더라도 poGM은 압축된 형태의 그래프를 해석하는 비용이 들기 때문에, 기억 장소 재활용 체계를 한 번 수행하는 시간은 약간 늘어난다. 그러므로, 기억 장소 재활용 체계의 수가 줄어든 만큼의 속도 향상은 되지 않았다.

ZGM과 비교해 보면, 적은 힙 공간에서 poGM이 기억 장소 재활용 체계의 수행 횟수가 크게 줄어듬을 알 수 있다. 그리고, 주어진 힙 공간이 커짐에 따라서 감소한 기억 장소 재활용 체계의 비율이 10% 줄었다. 이때 수행 속도는 3% 정도 밖에 감소하지 않았다. 이것은 주어진 힙 크기가 큰 구간에서 수행 시간은 기억 장소 재활용 체계를 위한 시간보다는 실제 수행 시간에 영향을 받기 때문이다.

그러므로, 본 실험을 통하여 poGM은 기억 장소 재활용시 GM보다 힙 공간을 효율적으로 활용한다는 것을 알 수 있다.

5. 결 론

본 논문에서는 poGM을 설계하고 정의하였으며 시뮬레이션을 실험을 통하여 GM,ZGM과 성능을 비교하였다. poGM은 GM의 절대 주소나 우회 노드에 자료를 중복 사용함으로써 공간 효율을 높일 수 있도록 고안된 추상기계로서, GM의 힙 크기를 줄인 ZGM에서 발생하는 자료의 태그 옮김이 일어나지 않는다.

실험을 통하여 컴파일된 코드를 동일한 힙 크기에 대하여 수행 시간을 비교한 결과 poGM은 GM보다 평균 30%, ZGM보다 평균 6.5% 정도 poGM의 수행 속도가 증가하였다. 컴파일된 코드의 힙 사용을 비교한 결과 poGM은 총 힙 사용량의 경우 GM보다 평균 32.0%, ZGM보다 평균 4.7% 정도의 공간을 절약할 수 있었다. 또한 최소 힙 사용량의 경우 GM보다 평균 47.6%, ZGM보다 평균 22.6 % 정도의 힙 공간이 적은 경우에도 프로그램 수행이 가능하다는 것을 보임으로서 힙 사용량을 적게 사용한다는 것을 보였다.

전체적으로 실험 결과를 통해서 poGM은 GM보다 총 힙 사용량과 최소 힙 사용량을 크게 줄일 수 있으며, 동일하게 힙 크기가 주어졌을 경우에도 수행 시간을 줄일 수 있다는 것을 알 수 있다. 또한 공간 효율을 줄일 ZGM과 비교하여서도 힙 사용량과 수행 시간을 줄일 수 있음을 알 수 있다.

참 고 문 헌

[1] J. Backus. "Can Programming Be Liberated form the von Neumann Style? A Functional Style and Its Algebra of Programs". *Communications of the ACM*, 21(8):613-641, August 1978.

[2] J. Hughes. "Why Functional Programming Matters". In D. A. Turner, editor, *Research topics in Functional Programming*, UT Year of Programming Series, chapter 2, pp.17-42, Addison-Wesley, 1990.

[3] L. Augustsson. "A Compiler for Lazy ML". In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pp.218-227, 1984.

[4] T. Johnsson. "Efficient Compilation of Lazy Evaluation". In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pp.58-69, 1984.

[5] Gyun Woo. *A Space-Efficient G-machine Using Tag-Forwarding*. PhD thesis. Korea Advanced Institute of Science and Technology, Taejon, Republic of Korea, 2000.

[6] Gyun Woo. A Non-recursive Garbage Collection for the ZG-machine, *프로그래밍언어연구회 추계 합동 학술 발표회 논문집*, pp.1-6 , September 2000.

[7] C. J. Cheney. "A Non-recursive List Compacting Algorithm". *Communications of the ACM*, 13(11): 677-678, November 1970.

[8] R. R. Fenichel and J. C. Yochelson. "A LISP Garbage Collector for Virtual Memory Computer Systems". *Communications of the ACM*, 12(11):611-612, November 1969.

[9] 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.

[10] T. Johnsson. "Target Code Generation from G-machine Code". In R. M. Keller and J.H.Fasel, editor, *Proceedings of a Workshop on Graph*

Reduction, volume 279 of Lecture Notes in Computer Science, pp.119-159, Sante Fé, NM, 29th September-1st October 1986. Springer. 1986

[11] W. Partain. "The nofib Benchmark Suite of Haskell Programs". In J. Launchbury and P. M. Samson, editors, *Functional Programming, Glasgow, Workshops in Computing*, pp.195-202, Springer Verlag, 1992.

박 홍 영

1993년 한국과학기술원 전산학과 학부 졸업. 1995년 한국과학기술원 전산학과 석사 졸업. 현재 한국과학기술원 전산학과 박사 과정. 현재 인공위성연구센터 연구원. 관심분야는 함수형 언어, 인공위성 관리 S/W

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