포인터와 자료를 중복하여 공간 효율을 높인 GM

박홍영[U] 한태숙
한 국 과 학 기 술 원 전자/전산학과 전산학전공
{hypark,han}@pllab.kaist.ac.kr

A space-efficient GM using pointer-overlapping

Hong-Young Park[U] Tai-Sook Han
Dept. of EE & CS, KAIST

요 약

G-Machine은 수행하는데 있어서 그래프를 표현하기 위해 많은 그래프 공간을 필요로 한다. 이를 줄이기 위한 노력으로 최근 태그 옮김을 이용한 ZGM이 설계되었다. 하지만, ZGM은 태그와 자료의 분리로 인하여 많은 상대 주소를 갖게 되어 실행 시간 부담이나 공간 활용 부담이 된다. 본 논문에서 우회 노드 장소에 자료를 중복 사용하는 방법을 이용하여 G-Machine에서 필요로 하는 그래프 공간을 절약한 poGM(Pointer Overlapped GM)을 제안한다. poGM은 ZGM과 같이 상대 주소를 사용하지만, 일반적인 노드 나타내는데 ZGM과 달리 자료의 분리가 일어나지 않고, 상대 주소를 줄일 수 있으며, 실험을 통하여 공간 효율과 시간 효율이 높은 것을 보인다.

1. 서론

지연 함수형 언어를 위한 추상 기계의 대부분은 그래프 축약(graph reduction)에 기반을 두고 있다. 특히, 컴파일 방식을 통해 빠른 그래프 축약을 지원하는 GM[1,2]은 현재까지도 지연 함수형 언어 구현 방법의 하나로 사용되고 있다. 지연형 함수형 언어를 그래프 축약으로 축약 할 경우 계산되지 않은 값은 그래프로 자연스럽게 나타낼 수 있고, 일반적으로 계산되지 않은 표현식을 나타내는 그래프는 그 표현식의 결과를 나타내는 그래프 보다 커다란 공간에 저장되므로, 이를 저장하기 위해 그래프 축약 기계는 많은 공간을 필요로 한다. 이러한 그래프 공간을 줄이기 위해 ZGM[34]은 특별한 부호화와 태그 옮김(tag forwarding) 통해 그래프의 저장 형태를 줄여 보다 작은 그래프 공간에 저장할 수 있게 하였다. 그러나, ZGM에서 모든 노드를 태그 옮김으로 인해 태그와 자료가 분리된다. 본 논문에서는 문제가 없이 각 노드에 대한 공간을 줄일 수 있는 방법을 제안하여, 그래프 공간을 줄이고, 실행 시간을 줄일 수 있게 한다.

2. 관련 연구

poGM의 비교 대상으로 GM과 ZGM을 비교 대상으로 하며, Woo[34]에서 기술한 내용을 기반으로 기술한다.

2.1. GM

스웨덴 Charmers 공대에서 Augustsson과 Johnsson에 의해 고안된 GM[1,2]은 그래프 축약을 이용하여 프로그램을 수행하기 위한 추상 기계이다. GM은 입력으로 조합자(combinator)프로그램을 갖는다. 주어진 프로그램의 각 완전 조합자는 컴파일 규칙에 위해 G-code라는 중간 표현으로 변환되고, 상태 변환 규칙에 의해 G-code를 수행함으로써 그래프 축약을 수행한다. GM의 그래프 축약에 의한 수행 과정은 각 완전 조합자에 대하여 몸체에 해당하는 그래프를 생성하는 과정과 이를 축약하는 과정으로 이루어진다. 이 과정은 좀 더 자세히 축약 가능 표현식 탐색, 스택에 인수 넣기, 그래프 생성, 축약 가능 표현식 갱신, 스택에서 인수 빼내기로 이루어지며, 더 이상 축약 가능한 표현식이 없을 때까지 반복 수행한다.

2.2. ZGM

ZGM은 GM의 그래프 생성시 그래프 저장 형태를 태그 옮김[4] 줄여 힙 공간을 절약할 수 있도록 한다. 태그 옮김은 GM에서 나타나는 우회 노드 부분에 자료의 태그와 상대 주소를 이용하여 힙 공간을 절약하는 방법이다. 하지만 모든 노드가 상대 주소를 갖아야 하고, 공유된 노드에 대해 태그 옮김을 사용할 수 없는 단점이 있다.

3. poGM

poGM도 그래프 저장 형태를 중복 기법을 이용하여 힙 공간을 절약할 수 있도록 제안한 추상기계다. poGM은 ZGM과 비교하여 태그와 데이터 공간의 분리를 막아서 컴파일 과정의 수행 시간도 적고, 컴파일 결과로 나온 코드의 실행 공간과 시간도 적고, 실행 시간에 압축된 그래프를 해석하는 부담 비용도 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

표 1. 입력 언어 정의

표[1]은 입력 언어를 정의한다. p는 입력 프로그램, d는 완전 조합자, e는 표현식, f는 완전 조합자 이름, x는 완전 조합자의 인수(또는 변수), i는 정수 값이다. 또한 프로그램 전체를 통하여 같은 이름을 갖는 완전 조합자는 없다고 가정한다.

3.2. 그래프 표현 형식

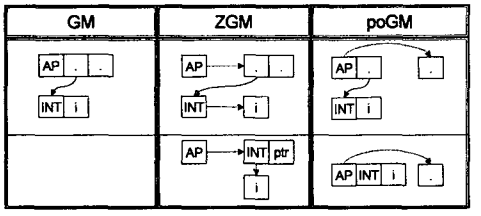


그림 1. 추상 기계에 따른 그래프 표현

그림[1]은 적용 노드의 왼쪽은 정수 표현, 오른쪽은 다른 우회 노드가 있는 상태의 가능한 그래프 표현을 나타낸다. 윗부분은 압축이 일어나지 않는 형태, 아래부분은 압축이 일어난 상태를 표시한다. GM은 모두 우회 노드로 표현되고, ZGM은 태그 옮김 표현으로, poGM은 중복 표현으로 나타난다. 그림에서 poGM은 ZGM에서의 상대 주소가 적으므로 노드 자료를 이용하기 위한 부담이 준다. 또한 노드 분리가 없으므로 노드 자체에서 태그를 저장하고 남는 공간을 이용할 수 있다. 그러므로, poGM은 ZGM보다도 공간 효율을 높일 있다.

3.3. poGM의 코드 생성

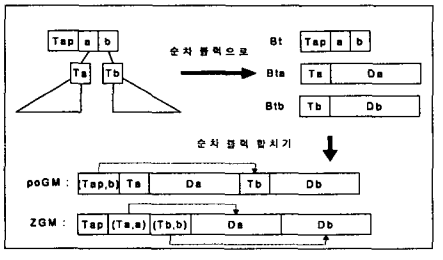


그림 2. 그래프를 압축하는 과정

그림[2]은 트리를 순차 블록의 형태로 바꾸는 과정을 나타낸다. 먼저, 트리의 일부분을 순차 블록 형태의 그래프로 변환한다, 각 블록 Bt, Bta와 Btb는 트리 Tap, Ta, Tb의 순차 블록 표현이다. 다음으로 나누어진 순차 블럭을 압축을 하면서 합친다. ZGM은 태그 옮김을 사용여, Bta에 있는 태그 Ta가 a영역으로 옮겨지고, Tb가 b영역으로 옮겨져 a가 (Ta,a)로, b가 (Tb,b)형태로 압축된다. poGM은 데이터 자체가 옮겨지는 것으로 (Tap,b)형태로 옮겨진다. 이것은 컴파일 과정에서 이루어지며, poGM에서 태그를 옮기는 절차가 필요 없음으로 컴파일시 성능 향상을 기대할 수 있다.

3.4. 명령어와 그래프 노드

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

(a) 명령어 (b) 노드 종류

표 2. poGM의 명령어와 그래프 노드 종류

poGM의 명령어는 ZGM과 호환을 이루는 형태로 정의했다. poGM의 그래프 노드 종류는 개별적인 워드 단위로 분리하였지만, 대부분의 노드의 구조는 해체되지 않는다. (AP,o)는 태그와 상대주소의 순서쌍을 나타낸다는 면에서 ZGM과 구조는 같지만 상대주소 o는 오른쪽 노드의 상대 주소를 나타낸다. (INT o)노드는 호환성을 위한 표시이고, 상대 주소가 없는 (INT)노드로 표현 가능하다. (FUN, *arity*)노드는 태그의 분리가 일어나지 않기 때문에 *arity*같이 한 워드를 모두 사용하지 않는 자료에 대해서 축약이 가능하다는 것을 나타낸다.

3.5. 컴파일 규칙

$\mathbf{F}[\![f\ x_1 \ldots x_k = e]\!]$	$= \mathbf{R}[\![e]\!][\ x_1=0, \ldots x_k=(k-1)\ ]\ k$
$\mathbf{R}[\![e]\!]\ \rho\ k$	$= \mathbf{C}[\![e]\!]\ \rho$ ++[UPDATE k, POP k, UNWIND]
$\mathbf{C}[\![e]\!]\ \rho$	$=[$ ALLOCSEQ ($\mathbf{N}[\![e]\!]\ \rho$) $]$
$\mathbf{N}[\![f]\!]\ \rho$	$=[$ PUSHFUN $f\ ]$
$\mathbf{N}[\![x]\!]\ \rho$	$=[$ PUSH ($\rho\ x$) $]$
$\mathbf{N}[\![i]\!]\ \rho$	$=[$ COPY (INT) , COPY $i\ ]$
$\mathbf{N}[\![e_0\ e_1]\!]\ \rho$	$=[$ COPY (AP,o)$]$ ++ $\mathbf{N}[\![e_0]\!]\ \rho$ ++ $\mathbf{N}[\![e_1]\!]\ \rho$
	where $o = \lvert tl_0 \rvert + 1$

표 3. poGM 의 컴파일 규칙

표[3]은 입력 언어를 G-code로 컴파일 하는 규칙이다. poGM의 컴파일 규칙은 GM과 ZGM에서 사용하는 컴파일 규칙과 유사하게 완전 조합자 하나를 컴파일 하기 위한 **F**, 완전 조합자의 반환값을 위한 **R**, 그래프를 생성하기 위한 **C**, **N**규칙으로 구성된다. 특히 poGM에서 **C**는 ZGM에 있는 상대 주소는 필요 없고, **N**은 **C**로부터 표현식 e와 변수 환경 ρ 만을 전달받는다는 것이 ZGM과 다르다.

3.6. 상태 변환 규칙

poGM의 상태 변환 규칙 표[10]에 의해 정의되었다. poGM의 상태는 명령어 리스트, 스택, 그래프 저장소, 완전 조합자 참조 리스트의 4가지 원소로 구성한다. poGM은 초기 상태< c_{init},[],G_{init},E_{init} >를 수행하여 결과값을 얻는다. 초기 명령어 목록 c_{init} 은 [ALLOCSEQ [PUSHFUN *main*], UNWIND]이다. ALLOCSEQ, PUSHFUN, PUSH, COPY, DONE, UPDATE, POP등 대부분 명령어들은 ZGM과 같지만, UNWIND는 노드에 따라 처리하는 방법이 다르며, 이때 노드를 찾고, 함수 수행시 실행 연산이 감소한다.

4. 실험 및 분석

각 추상 기계는 입력 언어를 C언어로 번역하고, 실행 시

1. <	ALLOCSEQ c' : c,		s, G	, E	>
⇒ <	c' : DONE : c,		$0{:}n$: $s, G[n=\ alloc(\|c'\|)]$	, 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	>

표 4. poGM 의 상태 변환 규칙

간 시스템과 함께 컴파일하여 수행 코드를 생성한다. 정확한 비교를 위하여 각 기계는 유사한 구조로 설계하여, 많은 부분을 공유하였다. 실험 환경으로 PII-466, 128M RAM, LINUX(mizi0S 1.0), GNU C(egcs-2.91.66) -O2 옵션을 사용하였다. 벤치마크로 하스켈 nofib 벤치마크 프로그램 (imaginary subset) 몇 개를 이용하였다.

기억 장소 재활용 체계는 연속된 공간을 확보할 수 있는 복사 알고리즘을 사용하였다. GM은 너비 우선 방식, ZGM, poGM은 깊이 우선 방식을 이용하였다.

평가 기준은 사용하는 총 힙 사용량, 수행할 수 있는 최소 힙 사용량을 측정하였으며, 같은 힙 크기에서 수행 속도를 측정하였다.

4.1. 힙 사용량

	exp	nfib	primes	queens	tak
(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 5. 총 힙 사용량

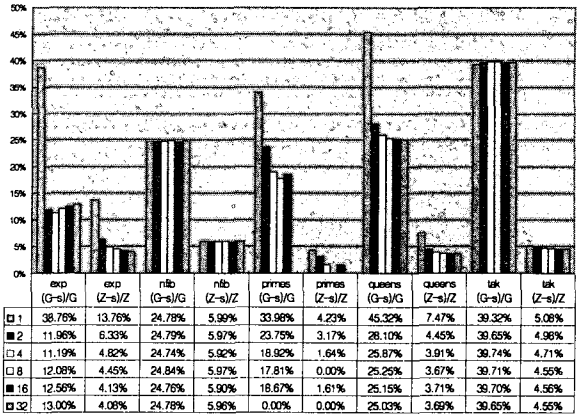
표[5]은 총 힙 사용량을 비율로 나타낸다. 첫째 행은 각 프로그램을 표시하고, 첫 열은 각 기계에 따른 수행 향상 비율을 나타낸다. 실험 결과 poGM이 GM보다 30%, ZGM보다도 1 에서 9 % 정도 줄었다.

	exp	nfib	primes	quuens	tak
(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%

표 6. 최소 힙 사용량

표[6]는 프로그램을 수행하는데 필요한 최소 힙 워드의 크기를 나타낸다. poGM은 GM보다 31 ~ 55 %, ZGM보다도 7 ~ 33 % 정도 줄었음을 알 수 있다. 이는 메모리 공간에 제약이 있는 시스템에서 더욱 효율적으로 사용 가능함을 의미한다. 실험 결과 poGM이 공간 효율적이다.

4.2. 동일한 힙 공간이 주어진 경우



	exp (G-s)/G	exp (Z-s)/Z	nfib (G-s)/G	nfib (Z-s)/Z	primes (G-s)/G	primes (Z-s)/Z	queens (G-s)/G	queens (Z-s)/Z	tak (G-s)/G	tak (Z-s)/Z
1	38.76%	13.76%	24.78%	5.99%	33.98%	4.23%	45.32%	7.47%	39.32%	5.08%
2	11.96%	6.33%	24.79%	5.97%	23.75%	3.17%	28.10%	4.45%	39.65%	4.96%
4	11.19%	4.82%	24.74%	5.92%	18.92%	1.64%	25.87%	3.91%	39.74%	4.71%
8	12.08%	4.45%	24.84%	5.97%	17.81%	0.00%	25.25%	3.67%	39.71%	4.55%
16	12.56%	4.13%	24.76%	5.90%	18.67%	1.61%	25.15%	3.71%	39.70%	4.56%
32	13.00%	4.08%	24.78%	5.96%	0.00%	0.00%	25.03%	3.69%	39.65%	4.55%

그림 3. 동일한 힙에 따른 수행 시간 변화율

그림[3]은 동일한 힙 크기에 대한 수행 시간 효율을 나타낸다. 숫자 1은 단위로 실험에 사용된 입력 프로그램의 각 추상 기계에서 사용하는 최소 힙 중에서 가장 큰 것으로 하였다. 그림에서 힙 공간이 1 인 경우, 기억 장소 재활용 체계의 호출 횟수에 의해 최고 45%정도로 높은 속도 향상을 보인다. 그러나, 일반적으로 GM보다 10 ~ 40 % 정도, ZGM보다는 5% 속도가 향상되었다. 본 실험 결과 수행 시간 면에서도 GM이나 ZGM보다 효율적이다.

5. 결론

본 연구에서 제안한 poGM은 자료 중복을 사용하여 공간 효율을 높인 추상 기계이며, 실험 결과 poGM은 다른 기계에 비하여 사용 공간 효율과 수행 시간 효율이 높았다.

6. 참고 문헌

[1] L. Augustsson. "A Compiler for LazyML". In proceedings of the ACM Symposium on Lisp and Functional Programming, pages 218-227, 1984.

[2] T. Johnsson. "Efficient Compilation of Lazy Evaluation". In Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, page58-69, 1984.

[3] 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.

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