제 18 회 한국정보처리학회 추계학술발표대회 논문집 제 9 권 제 2 호 (2002.11)

G-machine 에서의 AP 노드 재사용

주성용*, 우 균**
동아대학교 컴퓨터공학과

e-mail : jheaven1@donga.ac.kr*, woogyun@mail.donga.ac.kr**

Reusing AP Node in G-machine

Sung-Yong Joo*, Gyun Woo**
Dept. of Computer Engineering, Dong-A University

요 약

G-machine은 지연 함수형 언어를 효율적으로 구현하기 위한 추상기계이다. G-machine은 축약 과정에서 생성되는 그래프를 저장하기 위해서 많은 기억장소를 필요로 한다. 본 논문은 힙에 할당되는 그래프 노드 중 AP 노드를 재사용하는 방법을 제시한다. 일반적으로 AP 노드를 루트로 하는 그래프가 한 단계 축약된 후 다시 AP 노드를 루트로 하는 그래프로 구성되어지는 경우 루트 노드를 재사용할 수 있다. 이를 위해서는 기존 G-machine을 약간 확장 시켜야 하는데, 본 논문에서는 AP 노드의 일부를 변경하기 위한 명령어 UPDL 과 UPDR을 제시하고, 이들 명령어의 상태변환 규칙과 이들 명령어 생성을 위해 수정된 R-변환 규칙을 제시한다. 본 논문에서 제시하는 방식으로 기 생성된 AP 노드를 재사용함으로써 힙 기억공간의 사용량을 줄일 수 있고, 이 과정에서 힙 기억장소 할당과 쓰기 연산의 수를 줄일 수 있다.

1. 소개

Johnsson은 지연 함수형 언어(lazy functional languages)를 효율적으로 구현하기 위한 한 방법으로 G-machine[1]을 제안하였다. G-machine은 입력 프로그램으로 완전조합자(supercombinators)를 받고, 각 완전조합자를 G-machine 명령어로 반환한다. 변환된 명령어는 수행 시간에 완전조합자의 몸체에 해당하는 그래프를 생성하고, 생성된 그래프를 축약한다. 일반적으로 이러한 축약 과정중에 생성되는 그래프는 힙 기억장소에 저장된다.

이 논문은 G-machine 그래프 축약과정중 AP 노드 재사용[5]에 관한 논문이다. 축약 전 그래프의 루트가 AP 노드이고 축약 후 그래프의 루트도 여전히 AP 노드인 경우 기존 AP 루트를 재사용할 수 있다. 이 논문에서는 이렇게 AP 노드를 재사용할 수 있는 경우를 기술하고 이를 위한 명령어, 컴파일 규칙, 상태변환 규칙을 제시한다.

이와 같이 특수한 경우에 기억공간을 줄이기 위한 시도로서 Mark P. Jones가 제시한 루트 최적화(root optimization)[3]가 있다. 이 방법은 함수 재귀 호출 시, 주어진 초기 인수 그대로 함수 몸체에서 호출될 때 적용된다. 이러한 경우 축약 후 그래프의 일부가 축약 전 그래프와 같은 형태가 되는데, 루트 최적화는 이 경우 기존 그래프를 재사용하도록 한다. 루트 최적화는 실제 컴파일러에서 성능평가된 바[4] 있다.

이 논문에서 제시된 방법도 새로운 AP 노드를 만드는 대신 이미 존재하는 AP 노드를 활용한다는 점에서 루트 최적화와 유사하다. 그러나 재귀 호출 함수뿐만 아니라 일반적인 함수에도 적용될 수 있다는 점에서 루트 최적화와 다르다. 본 논문의 방법은 AP 노드를 루트 노드로 하는 그래프를 축약한 결과 그래프도 AP 노드를 루트 노드로 할 경우, 해당 함수의 재귀호출 여부와 상관 없이 적용가능하다.

이 논문의 구조는 다음과 같다. 먼저, 2장에서 기존 G-machine에서의 그래프 축약 방법을 제시하고, 3장에서 AP 노드 재사용 G-machine에서의 상태 변환 규칙과 추가된 명령어를 기술하겠다. 그리고 4장에서 AP 노드 재사용 G-machine에 추가된 명령어와 기존 G-machine의 명령어 성능을 비교하고, 끝으로 5장에서

결론을 맺도록 하겠다.

2. G-machine에서의 그래프 축약

G-machine에서는 임의의 표현식(expression)을 힙 공간의 그래프로 표현한다. G-machine에서의 프로그램 수행은, 값을 구하고자 하는 표현식에 해당하는 그래프를 더 이상 축약할 수 없을 때까지 계속해서 반복적으로 축약해 나가는 과정이다.

G-machine에서의 그래프 축약 과정을 보기 위해 함수합성을 수행하는 완전 조합자 comp를 생각해보자. 그림 1은 함수 합성을 위한 완전조합자 comp의 정의와 G-machine 명령어를 각각 보인 것이다. 그림 1-(나)는 그림 1-(가)의 완전조합자의 정의를, Peyton Jones와 Lester가 정리한 G-machine[2]의 컴파일 규칙에 따라서 번역한 결과이다. 완전조합자 comp에 대한 그래프 축약과정을 보이면 그림 2와 같다.

```
comp f g x = f (g x)
```

```
PUSH 2;
PUSH 2;
MKAP;
PUSH 1;
MKAP;
UPDATE 3;
POP 3;
UNWIND;
```

(가) 완전조합자 comp의 정의 (나) comp의 G-code

그림 1. comp의 정의와 G-code

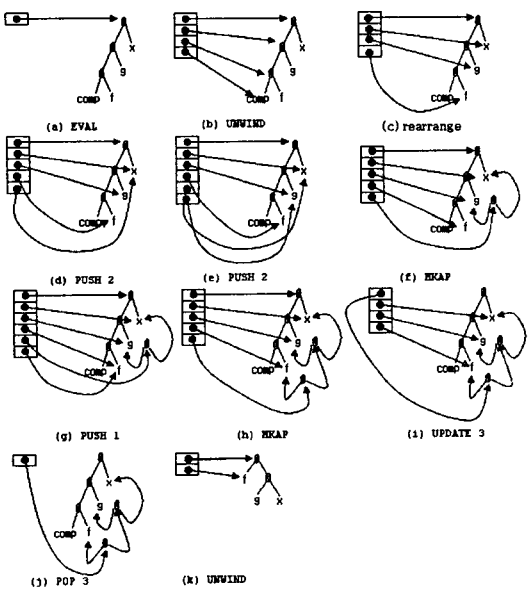

그림 2. G-machine에서 그래프 축약

그림 2는 comp f g x가 그림 1-(나)의 명령어 순서에 따라서 축약되는 과정을 나타낸 것이다. 그림 2에서 (a), (b), (c)는 그림 1의-(나)에서 찾아볼 수 없는 부분이다. 이 세 개 과정은 완전조합자 comp를 호출한 프로그램의 다른 부분에 의해서 실행되어진 것이다. EVAL은 현재 스택과 명령어를 덤프라 불리는 다른 스택으로 나머지 명령어 순서와 스택 상태를 저장하고, 새로운 스택을 생성하는 명령이다. 그리고, rearrange는 G-code 명령어는 아니다. 이것은 AP 노드의 함수부에 적용될 인수부를 가리키도록 그래프를 조정하는 것이다.

그림 2의 과정 (f)와 (h)에서 사용된 명령어 MKAP는 스택의 최상위 원소와 다음 원소를 취해서 새로운 AP 노드를 만들고, AP 노드를 만들기 위해서 사용했던 스택의 두 원소를 제거한 다음 스택의 최상위에 새로 생성된 AP 노드 포인터를 삽입하는 명령어이다. UPDATE k는 스택의 최상위로부터 (k+1)번째 위치한 원소를 스택의 최상위 원소로 갱신한 후 스택 최상위 원소를 스택으로부터 제거하는 명령어이다.

그림 2에서 축약 전 그래프의 루트는 그림 2-(h)에서 생성된 AP 노드로 갱신된다. 그러나 갱신 후 그래프의 루트도 여전히 AP 노드이므로 새로운 AP 노드를 생성하는 대신 기존 AP 루트의 함수 포인터와 인수 포인터를 변경함으로써 같은 효과를 얻을 수 있다.

3. AP 노드 재사용

G-machine에서 사용하는 노드 중 본 논문에서 재사용하고자 하는 AP 노드의 구조를 나타내면 그림 3과 같다. 그림 3에서 n_0는 적용할 함수를 가리키는 포인터이고, n_1은 n_0가 가리키는 함수에 적용될 인수를 가리키는 포인터이다.

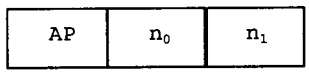

그림 3. G-machine에서의 AP 노드 구조

본 논문에서는 AP 노드를 새로 할당하는 대신 기생성된 AP 노드를 활용한다. 이를 위해서는 AP 노드의 두 포인터 n_0와 n_1을 직접 갱신하기 위한 새로운 명령어가 필요하다. 본 논문에서는 두 개의 명령어 UPDL(update left pointer), UPDR(update right pointer)를 추가하고자 한다. UPDL과 UPDR의 의미는 그림 4와 같다.

그림 4는 새로운 두 명령어 UPDL k과 UPDR k의

상태 변환 규칙이다. UPDL k는 스택의 최상위 원소를 0으로해서 스택의 k번째 원소가 가리키는 함수부 포인터를 스택의 최상위 AP 노드의 포인터로 갱신한다. 마찬가지로, UPDR k는 스택의 k번째 원소가 가리키는 AP 노드의 인수부 포인터를 스택의 최상위 원소가 가리키는 노드가 되도록 갱신한다.

그림 5는 새롭게 정의한 두 명령어를 생성하기 위해서 기존 G-machine의 R-변환규칙을 변경한 것이다. R-변환규칙의 ρ는 완전조합자의 인자 정보를 제공하는 함수이다. 이 규칙은 표현식 e_0가 표현식 e_1에 적용되는 형태에 대하여, 즉 AP 노드를 만들어야 하는 경우에 대하여 규정하고 있다. R-변환규칙 내에 사용된 C-변환규칙은 기존 G-machine의 변환규칙과 같다.

$$\langle \mathbf{UPDL}\ k{:}\ i,\ a_0 : a_1 : \ldots : a_k : s,\ h[a_k = \mathbf{AP}\ n_0 n_1],\ m \rangle$$
$$\Rightarrow \langle i,\ a_1 : \ldots : a_k : s,\ h[a_k = \mathbf{AP}\ a_0 n_1],\ m \rangle$$

$$\langle \mathbf{UPDR}\ k{:}\ i,\ a_0 : a_1 : \ldots : a_k : s,\ h[a_k = \mathbf{AP}\ n_0 n_1],\ m \rangle$$
$$\Rightarrow \langle i,\ a_1 : \ldots : a_k : s,\ h[a_k = \mathbf{AP}\ n_0 a_0],\ m \rangle$$

그림 4. 상태 변환 규칙

$$R[\![e_0\ e_1]\!]\rho\ k =$$
$$C[\![e_1]\!]\rho{+}{+}\ \mathbf{UPDR}\ k{+}{+}C[\![e_0]\!]\rho{+}{+}[\mathbf{UPDL}\ k, \mathbf{POP}\ k, \mathbf{UNWIND}]$$
ρ: function
k: arity

그림 5. R-컴파일 규칙

그림 6은 그림 5의 컴파일 규칙에 따라서 완전조합자 comp를 컴파일한 결과이다. 그림 1-(나)와 비교해 보면 동일한 개수의 G-machine 명령어로 번역되었음을 알 수 있다. 첫번째 MKAP 다음 UPDR 3 명령어가 사용되었고, 두번째 MKAP와 UPDATE가 UPDL 3 명령어로 대체되었다.

```
comp:PUSH 2;
     PUSH 2;
     MKAP;
     UPDR 3;
     PUSH 0;
     UPDL 3;
     POP 3;
     UNWIND;
```

그림 6. 새로운 G-code 스트림

루트 최적화와 비교해보면, 루트 최적화의 경우에 완전조합자 정의에서 재귀호출을 하는 경우 정적인자가 항상 재귀호출의 처음 부분에 나타나는 경우에 적용되었다. 그러나, 본 논문에서 제시된 방법은 그와는 상이하게 G-machine 명령어의 마지막에서 MKAP와 UPDATE의 조합으로 나타나는 경우에 적용된다.

그림 7의 과정 (g)와 (i)는 그림 4의 상태 변환 규칙에 따라 항상 조합으로 사용되어진다. 이 두 과정은 기존 G-machine의 MKAP와 UPDATE 3을 대신한다. 그림 7-(g)는 그림 4의 규칙에 따라 그림 7-(f)의 4번째 원소가 가리키는 AP 노드의 인수부 포인터를 최상위 원소가 가리키는 원소를 가리키도록 구성한 것이다. 다음 과정으로 스택의 최상위 원소를 제거한다. 그림 7-(h)에서는 스택의 4번째 원소가 가리키는 AP 노드의 함수부 포인터를 스택의 최상위 원소가 가리키는 원소로 변경하는 것을 제외하면 앞의 과정과 동일하다.

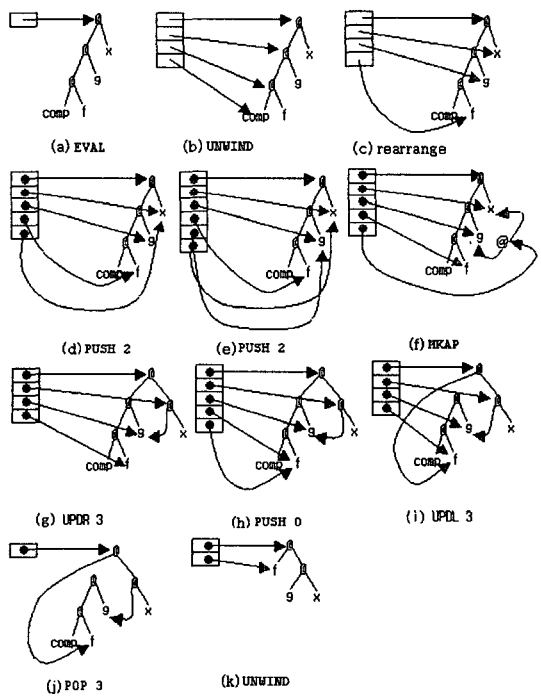

그림 7. AP 노드를 재사용하는 G-machine의 그래프 축약

4. G-machine과 AP 노드 재사용 G-machine의 비교

3장에서 AP 노드를 재사용하기 위한 컴파일 규칙과 두 명령어 UPDL, UPDR를 정의하였다. 축약 전 표현식의 그래프 루트가 AP 노드이고 축약 후의 결과 그래프의 루트가 다시 AP인 경우에는 축약 전 그래프

의 루트 노드를 재사용할 수 있다. 이러한 특수한 경우에 AP 노드를 재사용함으로써 G-machine 명령어 MKAP를 적게 사용하게 되고, 그 결과로써 새로운 힙 기억공간을 할당하지 않게 됨으로써, 힙 공간을 절약할 수 있다. 또한 이 과정에서 G-machine 명령어를 실제로 수행하기 위한 여분의 명령어를 줄일 수 있다.

본 논문의 방법은 원래 G-machine 명령어 순서에서 마지막 부분에서 MKAP와 UPDATE k로 나타난다면, 이 부분을 새로 정의한 명령어 UPDL k와 UPDR k의 조합으로 대체하게 된다. 결국 MKAP-UPDATE를 UPDR-UPDL로 바꾼 결과가 이들 명령어 조합의 성능을 비교하면 표1과 같다.

표 1은 기존 G-machine과 AP 재사용 G-machine에서의 쓰기 연산과 새로 할당되는 메모리 워드의 수를 비교한 것이다. 여기서는 하나의 AP 노드는 세 개의 워드로 구성된다고 가정했다.또한 쓰기 연산은 워드 단위로 이루어진다고 가정했다.

MKAP의 경우 기억공간 할당과 할당된 기억공간에 쓰기 연산을 요구하므로, 매번 수행 시마다 3개의 워드를 생성한다. 또한 MKAP 연산은 AP 노드가 나타낼 함수부에 대한 포인터와 그 함수부에 적용될 인수에 대한 포인터를 기록하고 노드의 첫번째 워드에 노드의 종류를 기록해야 하므로 3번의 쓰기 연산을 수행해야한다. UPDATE의 경우도 태그의 종류를 나타내기 위한 가장 왼쪽의 워드와 함수부와 인수부를 새로 작성해야 하므로 3번의 쓰기 연산이 필요하다. 그러므로, 결과적으로 이 과정을 UPDL k와 UPDR k 명령어 조합으로 대체하게 되면 표 1과 같이 기억공간과 속도 향상을 기대할 수 있다.

표 1. GM과 AP Reusing GM의 명령어의 비교

	number of write operations	number of newly allocated words
G-machine (MKAP-UPDATE)	6	3
AP-Reusing GM (UPDR-UPDL)	2	0

또한 MKAP 명령에 의해서 쓰기연산 외에도 메모리에 새로운 3개의 워드를 할당하기 위한 연산도 요구된다. 본 논문의 방법을 적용할 경우 메모리 공간뿐만 아니라, MKAP에서 필요한 기억공간 할당 과정과 할당된 기억공간에 쓰기 연산의 회수를 줄일 수 있기 때문에 속도 향상도 기대된다.

5. 결론

이 논문에서 제안한 방법은 축약 전과 후의 그래프 루트가 모두 AP 노드인 경우로 한정된다. 그러므로, 제안된 방법을 이용해서 프로그램을 컴파일할 경우에 얻을 수 있는 성능향상은 프로그램 내에 AP 노드에서 AP 노드로의 그래프 축약이 얼마나 많이 포함되어있는지에 비례하게 된다.

이 논문에서 제안한 방법의 가장 큰 장점은 그래프의 AP 노드의 포인터을 바꿈으로써 기억공간 절약과 속도향상을 모두 얻을 수 있다는 점이다. 또한 이 방법은 매우 간단하기 때문에 이 방법을 실제 컴파일러에 적용한다고 해도 그렇게 큰 추가비용이 발생하지 않을 것으로 예상된다. 이러한 방법을 AP 노드뿐만 아니라 다른 노드로도 확장한다면 더 많은 성능향상이 있을 것으로 생각되는데 이는 향후 연구로 남겨둔다.

참고문헌

[1] Thomas Johnsson, "Efficienct compilation and lazy evaluation", *In Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 58-69, 1984

[2] Simon L. Peytone Jones and David R. Lester, *Implementing Functional Languages: a tutorial*, Prentice Hall, 1991.

[3] Mark P. Jones, *Implementation of the Gofer functional programming system*,ACM SIGPLAN, June 1993.

[4] Kwanghoon Choi, Chiung O, and Taisook Han, "A Practical Impremention of Root Optimization in G-machine", In Proceedings of Implementation of Functional Lanugages (IFL '98), 1998.

[5] 우균, "G-machine에서 AP 노드 재사용 방법", Technical Report CE 2002-01, 컴퓨터공학과, 동아대학교, 2002년 9월