탐색시간의 개선을 위한 함수형 배열의 설계 및 구현

주 형 석† · 유 원 희††

요 약

순수 함수언어는 참조적 투명성을 가지고 있으므로 모든 객체에 대한 직접적인 갱신이 불가능하다. 배열과 같은 집단자료구조가 갱신되면, 참조적 투명성을 유지하기 위하여 원래의 배열과 갱신된 배열이 모두 유지되어야 한다. 따라서 모든 자료에 대한 참조적 투명성을 유지하면서 자료의 유지 비용을 줄일 수 있는 효과적인 방법의 개발이 요구되고 있다.

본 논문에서는 이러한 문제를 해결하기 위한 함수형 배열을 제시하고, 검증하고자 하였다. 이와 같은 검증을 위해서, 제안된 방법을 컴비네이터 그래프 감축기 상에서 구현하였다. 제안된 방법에서 배열 연산에 대한 탐색시간을 줄일 수 있었으며, 갱신 비용과 최근 버전에 대한 접근은 갱신된 자료를 실행시간에 재구성하는 과정 없이 상수시간에 가능하였다.

A Design and Implementation of Functional Array for Improvement of the Traversal Time

Hyung Seok Joo† · Weon Hee Yoo††

ABSTRACT

Pure functional languages have the referential transparency feature so that all objects cannot be updated destructively. Once an aggregated data structure such as array is updated, both the original and newly updated array must be preserved to maintain referential transparency. Thus, it is required to develop an efficient mechanism with which can reduce the maintenance cost while maintaining referential transparency for whole data.

This study is to suggest a functional array to solve the problem, and then test it. For that, the proposed mechanism was implemented on a combinator graph reduction machine. The result shows that proposed mechanism reduces traversal time for array operations. Also, updating all versions and accessing the recent version are achieved in constant time without reconstruction of updated data in execution time.

1. 서 론

순수 함수언어는 참조적 투명성(referential transparency), 고계함수, 지연평가, 묵시적 병행성 등의 특성을 지닌 언어로서 수식을 평가하는 순서에 상관없이 전체적인 결과가 항상 일정하다는 장점을 갖는다 [1, 3, 12]. 순수 함수언어에서 배열과 같은 집단 자료구조(aggregated data structure)가 갱신되면, 참조적 투명성을 보장하기 위하여 갱신 전의 자료와 갱신 후의 자료가 모두 유지되어야 하는 갱신문제(updating problem)가 발생한다. 즉, 프로그램 내에서 갱신이 발생될 때마다 전체 배열에 대한 자료의 복사를 수행하게 되므로 함수 프로그램의 수행성을 저하시키고 기

† 정 회 원:유한전문대학 전자계산과 부교수
†† 정 회 원:인하대학교 전자계산공학과 교수
논문접수:1996년 1월 30일, 심사완료:1996년 8월 8일

억장소에 대한 오버헤드를 크게 증가시킨다[6, 9].

갱신문제를 해결하기 위한 방법으로는 컴파일 시간 분석 방법[4, 8, 16], 함수언어의 의미를 제한하는 방법[10, 13, 19], 실행시간에 자료구조를 효율적으로 구축하는 방법[2, 5, 7] 등이 있다. 컴파일 시간 분석 방법이나 함수언어의 의미를 제한하는 방법은 자료구조의 갱신에 따른 복사문제에 대한 오버헤드를 줄일 수 있는 장점을 갖고 있으나 컴파일 시간에 알 수 있는 정보는 제한적이며, 최근에 갱신된 정보만을 참조하는 단일 쓰레드 접근(single threaded access)에 기초하고 있다. 따라서 함수언어의 특성을 침해하지 않으면서 참조적 투명성을 보장하기 위해 최근에 갱신된 자료 뿐만 아니라 갱신 전의 자료를 효율적으로 유지하기 위한 방안이 필요하다.

본 논문에서는 실행시간에 자료구조를 효율적으로 구축하여 참조적 투명성이 보장되며, 배열에 대한 연산을 처리할 때 소요되는 탐색시간과 복사되는 공간을 최소화하기 위한 함수형 배열을 설계하고 구현하고자 한다. 제안된 함수형 배열은 트리구조를 이용한 방법[7]에서 복사되는 공간의 낭비를 줄이고, Baker의 shallow 바인딩에 기초한 방법[2, 5]에서 다른 버전을 참조할 때, 갱신된 자료의 환경을 실행시간에 재구성하는 오버헤드를 개선하기 위한 것이다. 본 논문에서 최근에 갱신된 자료는 임의 접근이 가능한 공간에 구성하며, 갱신 전의 과거 버전에 대한 참조는 갱신 정보를 이용하여 탐색시간을 줄인다. 또한 제안된 방법과 다른 방법의 탐색시간을 비교하기 위해서 컴비네이터 그래프 감축기[14] 상에서 함수형 배열을 구현한 결과를 보인다.

2. 관련 연구

2.1 함수형 배열

함수언어에서는 재귀적 알고리즘에 의해 자연스러운 구성이 가능한 리스트 자료구조가 많이 사용되고 있으나 대부분의 명령형 언어에서는 임의의 원소에 대한 접근과 갱신이 상수시간에 가능하고, 갱신을 위한 추가적인 공간과 리스트에서 요구되는 포인터를 위한 기억공간을 절약할 수 있는 배열을 많이 사용하고 있다[2, 11].

따라서 상수시간에 자료를 갱신하고 참조할 수 있으며, 알고리즘의 기본이 되는 배열을 함수언어에 도입하기 위한 많은 연구가 진행되고 있다. 함수형 배열은 배열을 생성하고 처리하는 방법에 따라서 증가형 배열(incremental array)과 단일형 배열(monolithic array)로 구분할 수 있다. 증가형 배열은 다음과 같은 3가지 연산자를 갖는다[11].

- newarray(n): 크기가 n인 배열을 생성하는 연산자
- select(a, i): 배열 a에서 i번째 원소값을 반환하는 연산자
- update(a, i, x): 다음과 같은 새로운 배열 a′을 생성하는 연산자

$$\text{select}(a', j) = \text{select}(a, j) \quad \forall\, j \neq i$$
$$\text{select}(a', i) = x$$

증가형 배열은 갱신연산시 배열의 임의의 원소를 갱신할 수 있으므로 명령형 언어의 배열과 유사하지만 근본적인 문제점은 프로그램 내에서 갱신연산이 발생될 때마다 함수언어의 특성을 유지하기 위해, 그 배열과 같은 형태의 전체 배열을 매번 복사해야 하는 비효율성을 갖는다는 것이다. 예를 들어, 다음의 프로그램은 newarray(n)에 의해 생성된 배열 a에 대해 재귀적으로 init함수를 호출하여 각각의 원소를 x로 초기화하는 프로그램이다.

$$\text{init}(a, i, x) = \text{if } (i = 0) \text{ then } a \text{ else } \text{init}(\text{update}(a, i, x), i-1, x)$$

이때, init의 재귀적 호출은 배열 a와 같은 크기의 배열을 n번 복사하게 되며 각각의 복사에 대한 시간 및 공간복잡도는 $O(n^2)$이다. 따라서 배열 a를 초기화하는 데는 O(n2)만큼의 시간과 공간이 필요하게 된다. 이것은 동일한 프로그램이 명령형 언어의 배열에서 얻어지는 시간복잡도 O(n)과 공간복잡도 O(n)에 비해 상당히 비효율적이다.

함수형 배열의 다른 형태는 단일형 배열로서 배열 생성시 각 원소들을 특정값으로 초기화할 수 있는 배열이다. 이러한 배열을 생성하기 위해 정의될 수 있는 함수는 다양하다. 예를 들어, 함수 make_array(n, f)는 크기가 n이고 각 원소의 초기값은 filling 함수 f에 의해 결정된 배열을 생성한다. 즉, make_array(n, f)는

각각의 원소가 a[i]=f(i)로 초기화된 배열을 반환한다. 이와 같이 단일형 배열은 배열을 생성하고 초기화하는 과정에서 발생하였던 증가형 배열의 비효율적인 오버헤드를 줄일 수 있다[18]. 그러나 단일형 배열에서 갱신연산은 증가형 배열에서와 같이 filling 함수에 의해 초기값이 변경된 새로운 배열을 생성해야만 하는 문제점이 여전히 존재한다.

2.2 함수형 배열의 갱신문제

순수 함수언어에서 자료는 다른 프로그램에 의해 참조되어도 그 값이 변하지 않는 상수값이어야 하므로 모든 객체를 직접 갱신할 수는 없다. 따라서 배열을 사용하는 함수언어에서 각 원소에 대한 갱신을 수행할 경우 참조적 투명성을 보장하기 위한 갱신문제가 제기된다.

함수형 배열에 대한 갱신연산은 2.1절에서 서술한 바와 같이 갱신연산이 발생하는 횟수만큼 같은 형태의 배열을 매번 복사해야 하는 오버헤드를 갖는다[11]. 따라서 이와 같은 갱신문제를 해결하기 위한 방안으로 컴파일 시간 분석 방법[4, 8, 16], 함수언어의 의미를 제한하는 방법[10, 13, 19], 실행시간에 자료구조를 효율적으로 구축하는 방법[2, 5, 7] 등이 연구되고 있다.

컴파일 시간 분석 방법은 컴파일 시간에 함수형 언어에 대한 단일 쓰레드 접근을 분석하여 직접 배열의 원소를 파괴적으로 갱신(destructive update)하기 위한 방법으로 매우 효율적인 코드를 만들 수 있다는 장점을 갖는다. 그러나 이러한 분석 방법은 함수 프로그램의 수행이 특정 평가순서(evaluation order)에 의해서 이루어진다는 제한적인 가정에 따르고 있을 뿐만 아니라 컴파일 시간에 얻을 수 있는 정보가 한정적이라는 단점을 갖는다[5, 9]. 함수언어의 의미를 제한하는 방법은 상태를 표현하기 위한 구조를 함수언어의 구문에 추가하여 직접적인 갱신이 가능하도록 한 방법이지만, 단지 단일 쓰레드 방식의 프로그램만을 표현할 수 있도록 제한되어 있으며, 이를 나타내기 위한 형 시스템(type system)이 복잡해 진다[9, 10, 19, 20]. 이와 같은 두가지 부류의 방법들은 다중 쓰레드 접근을 위한 집단자료 구조를 허용하지 않으므로 함수언어의 표현력이 저하되며, 평가순서가 부분적으로 제한될 수 밖에 없는 문제점을 갖는다.

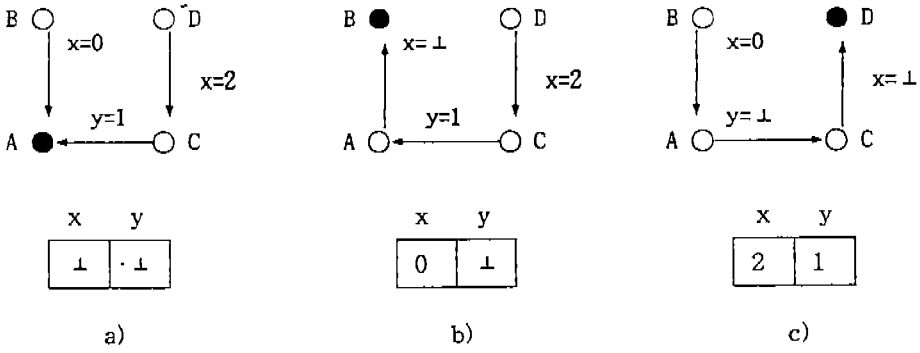
다중 쓰레드 방식인 경우는 갱신되기 전의 자료가 갱신 후에도 사용될 수 있으므로 갱신 전의 자료와 갱신 후의 자료가 모두 유지되어야 한다. 이와 같이 과거 버전에 대한 자료를 유지하기 위해서 실행시간에 자료구조를 효율적으로 관리하기 위한 방법들이 연구되고 있다. 이러한 방법의 대표적인 연구로는 배열의 인덱스를 트리를 이용하여 구현한 방법과 갱신된 자료만을 별도로 유지하는 shallow 바인딩 방법이 있다.

트리를 이용한 방법[7]은 배열을 이진 트리 형태로 구성하였는데 임의의 원소에 대한 참조나 갱신을 위한 탐색시간은 $O(\log_2 n)$이며, 배열의 복사를 위해서는 매번 탐색 경로 상에 존재하는 노드의 갯수만큼 새로운 노드를 할당받아야 하므로 기억공간 역시 $O(\log_2 n)$으로 상당히 큰 오버헤드를 갖게 된다. 특히 이 방법은 과거 버전에 대한 갱신이 불가능한 문제점을 갖고 있다.

갱신된 자료를 유지하기 위한 추가적인 기억공간과 갱신시간이 O(1)에 가능한 Baker의 shallow 바인딩에 기초한 방법[2, 5]들은 같은 버전에 대한 참조가 O(1)에 가능하도록 구성함으로써 같은 버전을 연속적으로 참조할 경우 커다란 장점을 갖는다. 그러나, 다른 버전의 참조가 필요한 경우 실행시간에 자료구조를 재구성해야 하는 단점을 갖는다. 특히 다른 버전에 대한 참조가 빈번하거나, 실행시간에 자료구조를 재구성할 때 갱신된 자료의 경로가 길다면, 자료구조를 재구성하는 비용이 매우 커지는 오버헤드를 갖는다. (그림 1)은 배열을 생성하고 배열의 원소를 다음과 같은 순서로 갱신하고 참조했을 때, shallow 바인딩 방법에서 자료를 유지하고 참조하는 방법에 대한 그림이다.

- A = Newarray(2); B = Update(A i_x 0); C = Update (A i_y 1); D = Update(C i_x 2);
- Select(B i_x);
- Select(D i_y);

(그림 1)에서 A, B, C, D는 각각의 갱신될 자료를 갖고 있는 환경을 나타내며, "●"으로 표시된 노드가 배열에 기억되어 있는 내용에 대한 현재 환경이다. (그림 1)의 a)는 처음 배열이 생성되고 이에 대한 갱



(그림 1) shallow 바인딩 방법의 예
(Fig. 1) Example of shallow binding scheme

신연산이 이루어진 상태의 그림으로서 배열에 기억된 내용은 배열 생성 당시의 값을 갖고 있으며 갱신할 내용은 버전 트리를 형성한다. b)는 환경 B의 상태를 참조하기 위해 환경을 재구성한 후의 그림이며, c)는 환경 D의 참조를 위해 재구성한 것이다. (그림 1)에서와 같이 shallow 바인딩 방법은 하나의 버전에서 다른 버전으로의 참조가 이루어질 때마다 환경을 재구성하는 비용이 커지는 문제점을 갖는다.

2.3 컴비네이터 그래프 감축기

함수언어의 구현을 위하여 이용되는 컴비네이터들 가운데 대표적인 것으로는 *SK* 컴비네이터[14, 15, 17]와 수퍼컴비네이터(super combinator)[3] 등이 있다. 본 논문의 구현에 사용된 *SK* 컴비네이터는 *S*, *K*, *I*라는 세가지의 컴비네이터를 기본으로 하며, 그 외에 컴비네이터 식의 최적화를 위한 *B*, *C*, *S'*, *B**, *C'* 컴비네이터, 재귀호출을 위한 *Y* 컴비네이터 및 산술, 논리, 리스트 등을 위한 원시 연산자들로 구성된다. 각 *SK* 컴비네이터의 감축규칙은 〈표 1〉과 같다.

〈표 1〉*SK* 컴비네이터의 감축규칙
〈Table 1〉 Reduction rule for *SK* combinator

R1.	$S\ f\ g\ x$	$\Rightarrow ((f\ x))\ (g\ x))$
R2.	$K\ x\ y$	$\Rightarrow x$
R3.	$I\ x$	$\Rightarrow x$
R4.	$B\ f\ g\ x$	$\Rightarrow (f\ (g\ x))$
R5.	$C\ f\ g\ x$	$\Rightarrow ((f\ x)\ g)$
R6.	$S'\ c\ f\ g\ x$	$\Rightarrow (c\ ((f\ x)\ (g\ x)))$
R7.	$B^*\ c\ f\ g\ x$	$\Rightarrow (c\ (f\ (g\ x)))$
R8.	$C'\ c\ f\ g\ x$	$\Rightarrow (c\ ((f\ x)\ g))$

본 논문에서 사용하는 그래프 감축 시스템은 크게 그래프 생성기, 그래프 감축기, 힙 기억장소와 가베지 수집기 등으로 구성되어 있으며 그래프 생성기에 의해 생성된 컴비네이터 프로그램 그래프의 각 노드들은 함수를 나타내는 셀과 그 함수에 대한 인수를 나타내는 셀로 구성된다. 각 노드들은 고정크기 노드로 구현되어 있으며 노드의 두 셀은 포인터 값이나 컴비네이터, 또는 정수값을 갖는다. 이러한 노드의 구성은 currying 형태로 표현된 컴비네이터 프로그램을 이진트리로 표현하는 것이 용이하며 기존의 태그를 사용하는 노드 형태보다 경우분석을 위한 수행 횟수가 적기 때문에 수행성이 높다[14]. 컴비네이터 프로그램 그래프의 감축은 이진트리로 표현된 그래프를 각 컴비네이터의 감축규칙에 따라 변환시켜 가는 과정을 의미한다. 또한 이 과정에서 컴비네이터들은 지연평가를 수행하므로 인수 부그래프에 대한 연산이 지금 즉시 필요하지 않다면 인수 부그래프에 대한 평가가 필요할 때까지 보류된다.

3. 함수형 배열의 설계

본 장에서는 함수형 배열의 설계 방안과 연산을 정의하며, 이를 구현하기 위한 자료구조를 설계하고, 이 자료구조를 이용하여 함수형 배열에 대한 수행 규칙을 기술한다.

3.1 함수형 배열의 설계 방안

함수형 배열의 자료구조를 지원하기 위한 기본연산은 배열을 생성, 갱신, 참조하는 연산이다. 명령형 언어에서 배열은 임의의 원소에 대해 참조 및 갱신이 O(1) 시간에 이루어지지만 함수언어에서는 참조적 투명성을 보장하기 위하여 갱신연산에 의해 변경된 과거 자료도 유지되야 하는 문제가 제기된다.

임의의 원소가 갱신될 때마다 나머지 자료를 모두 복사해서 별도의 버전을 유지하거나, 갱신된 정보를 트리를 이용하여 처리한다면 갱신과 참조를 위한 시간적, 공간적인 오버헤드가 커지게 된다. 이러한 문제를 개선하기 위해서 제안된 방법이 shallow 바인딩에 기초한 방법이다. 이 방법은 배열의 크기를 n이라고 하고 갱신된 자료의 갯수를 r이라고 할때, O(n+r)의 공간으로 함수형 배열을 유지하는 것으로 최적의 공간을 사용하며 같은 버전에 대해서 연속적인 참조를 수행할 때 매우 효율적이지만 다른 버전에 대한 참조가 행해지는 경우 환경을 재구성하는 비용이 커진다.

따라서 본 논문에서는 shallow 바인딩에서 환경을 재구성하는 비용을 줄이고 명령형 언어의 배열이 갖는 특성에 근접하기 위해서 다음과 같은 관점에서 함수형 배열을 설계한다.

- 배열의 최근 버전에 대한 참조시간을 O(1)에 가능하도록 구성한다.
- 임의의 원소에 대한 갱신은 추가적인 O(1) 공간으로 구성한다.
- 임의의 원소에 대한 갱신은 O(1)에 처리한다.
- 과거 버전에 대한 탐색시간을 줄인다.

배열을 이용한 많은 알고리즘이 가장 최근에 생성된 버전을 주로 참조하고 갱신되는 특성을 지니고 있으므로 최근 버전에 대해 갱신된 자료를 별도의 환경으로 구성하지 않고 임의 접근이 가능한 배열 공간에 나타낸다. 이와 같이 함으로써 최근 버전에 대한 자료는 O(1)에 참조가 가능하다. 이와 같은 설계 관점은 단일 쓰레드 접근으로 함수언어의 의미를 제한하여 파괴적으로 갱신하는 방법에서 착안된 것이다. 따라서 다중 쓰레드 방식의 처리가 가능하고, 함수언어의 표현력을 억제하지 않으면서도 최근 버전에 대한 효과적인 참조가 가능해 진다.

임의의 원소를 갱신하는 경우, 갱신을 위한 추가적인 공간은 O(1)이 되도록 하며, 이에 대한 갱신시간도 O(1)에 가능하도록 설계한다. 또한, 과거 버전에 대한 참조의 경우는 갱신연산에 의해 나타내진 갱신 정보를 이용하여 참조가 가능하며, 그렇지 않을 경우는 연결 리스트를 탐색하여 참조하도록 설계한다.

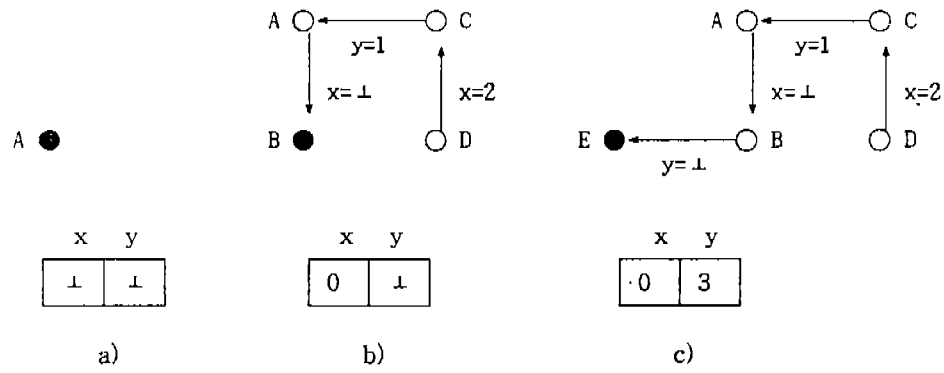
3.2 함수형 배열 연산자의 정의

본 절에서는 함수형 배열의 설계 방안에 따른 연산자들을 정의하고 연산 규칙을 설명한다.

배열 연산자는 각각 자신이 필요한 갯수의 인수를 취하는 함수로서 연산자 mka는 함수형 배열을 생성하는 함수이며, sel은 배열로부터 인덱스에 해당되는 값을 반환하는 함수이며, upd는 배열을 갱신하기 위한 함수로서 다음과 같이 정의한다.

- mka(n) : n개의 원소로 구성된 임의 접근이 가능한 원시배열을 생성
- upd(a, i, x) : 배열 a가 최근 버전이면 배열의 내용을 갱신하고 과거 버전인 경우는 새로운 갱신 노드를 생성하며, 갱신 정보를 reset
- sel(a, i) : 갱신 정보가 nil인 경우는 원시배열의 내용을 반환하며, 아니면, a의 연결 리스트로부터 인덱스 i에 대한 값을 반환

정의된 배열 연산자에서 n은 원시배열의 크기를 나타내며, a는 연산을 위한 배열을 지칭하며, i는 인덱스를 x는 갱신될 자료를 의미한다. mka 연산자는 임의 접근이 가능한 기억공간을 확보하고 각각의 갱신 정보를 nil로 초기화한다. upd 연산자는 자료를 갱신하기 위한 것으로 배열 a가 최근 버전일 경우 원시배열의 내용을 연결 리스트에 추가하고 원시배열의 내용을 갱신한다. 또한 과거 버전인 경우는 하나의 노드를 할당하여 갱신될 자료를 저장하여 과거 버전이 지칭하는 곳에 연결하여 정보를 유지한다. 이 때 원시배열의 해당 원소의 갱신 정보를 reset한다. sel 연산자는 배열의 i번째 값을 반환하기 위한 것으로 최근 버전인 경우는 원시배열의 내용을 O(1)에 반환하며, 과거 버전의 경우는 갱신 정보가 nil이면 원시배열의 정보를 반환하며, 그렇지 않은 경우는 해당되는



(그림 2) 제안된 함수형 배열의 예
(Fig. 2) Example of proposed functional array

배열의 연결 리스트로부터 인덱스 i의 값을 반환한다. (그림 2)는 본 논문에서 제안한 함수형 배열을 (그림 1)의 shallow 바인딩 방법과 비교하기 위한 그림이다.

(그림 2)의 a)는 처음 배열이 생성되었을 당시의 상태로서 원시배열의 최근 버전은 A이다. b)의 경우는 모든 원소가 갱신된 후의 상태로서 A 버전에 대한 갱신연산에 의한 B가 최근 버전이며 과거 버전 A에 대해 C와 D가 연결된 상태를 나타낸다. 이때 버전 B에 대한 참조는 환경을 재구성함이 없이 O(1)에 가능하며, 버전 D의 경우는 연결 리스트 상에 있는 자료를 통해 값을 반환한다. c)는 최근 버전 B에 대해 upd(B, iy, 3)의 갱신연산이 발생된 경우로서 최근 버전은 E가 되고, 갱신 자료는 원시배열에 유지되며 원시배열의 내용이 연결 리스트 상에 표현된다.

(그림 2)의 c)를 통해서 제안된 방법의 설계 방안을 검토해 보면, 최근 버전이 원시배열에 유지되어 있으므로 이에 대한 참조는 O(1)에 가능하며, 갱신을 위한 자료를 유지하기 위한 추가적인 공간은 O(1)로 구성되고, 갱신시간도 O(1)에 가능하다. 또한 과거 버전에 대한 탐색의 경우, 갱신되지 않은 원시배열의 자료는 상수시간에 가능하며, 갱신된 정보는 해당되는 버전의 위치로부터 탐색이 가능하기 때문에 환경을 재구성하는 시간에 비해서 부담이 줄어든다. 단지 초기 버전에 대한 갱신이 많아 질 경우에 갱신된 자료의 갯수를 r이라고 하면, 최악의 경우 O(r)의 탐색시간이 필요하다는 문제점을 갖는다. 〈표 2〉는 shallow 바인딩 방법과 제안된 방법의 함수형 배열을 주요 관점별로 비교한 것이다.

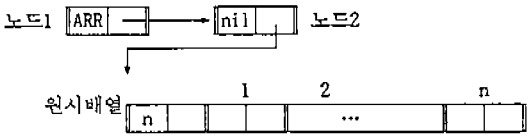
3.3 배열 연산을 위한 자료구조 및 감축규칙

본 절에서는 배열 연산을 위한 자료구조를 설계하고 연산자에 대한 컴비네이터를 정의하며, 설계된 자

〈표 2〉 함수형 배열의 비교
〈Table 2〉 Comparison of functional arrays

주요 관점	shallow 바인딩	제안된 방법
원시배열의 참조	O(1)	O(1)
최근 버전의 참조	원시배열이 동일 버전이 아닌 경우, 재구성 후 O(1)	O(1)
과거 버전의 참조	원시배열이 동일 버전이 아닌 경우, 재구성 후 O(1)	갱신 정보가 nil이면 O(1), 아니면 연결리스트 탐색
임의의 원소의 갱신시간	O(1)	O(1)
갱신에 의한 추가 공간	O(1)	O(1)

료구조를 이용하여 제안된 함수형 배열이 수행되는 규칙을 기술한다.



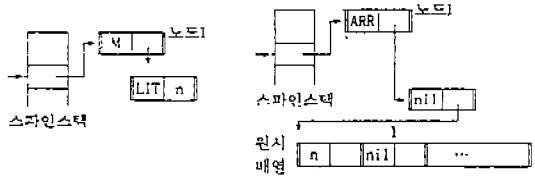
(그림 3) 함수형 배열의 자료구조
(Fig. 3) Data structure for functional array

본 논문에서 설계한 함수형 배열을 그래프 감축기상에서 구현하기 위한 내부적 자료구조는 (그림 3)과 같이 구성된다. (그림 3)은 1차원 함수형 배열에 대한 자료구조의 표현이지만, 배열의 그래프 구성 형태를 변경하여 2차원 이상의 다차원 배열로 확장할 수 있다. 노드1은 배열의 버전을 나타내기 위한 노드로서 왼쪽 셀은 자료형이 배열임을 나타내기 위한 형 컴비네이터 ARR을 가지며, 오른쪽 셀은 연결노드를 지칭한다. 노드2는 원시배열의 내용이 갱신될 경우 과거 자료를 유지하기 위한 연결노드로서, 왼쪽 셀은 갱신 자료의 끝을 나타내는 nil 값을 가지며 오른쪽 셀은 원시배열의 시작 위치를 갖는 포인터이다. 또한 원시배열은 임의 접근이 가능한 공간이며, 배열의 각 원소에서 왼쪽 셀은 갱신 정보를 가지며, 오른쪽 셀은 해당 원소의 값을 나타낸다.

(1) 생성 연산자 mka(n) : (M n)

임의 접근이 가능한 배열에 대한 생성은 (그림 4)와 같이 생성된다. (그림 4)의 a)는 컴비네이터 M을 수행하기 전의 그래프 형태이며 b)는 그래프 감축이 이루어진 후의 자료구조이다. 현재 스파인 스택의 포인터가 가리키는 노드의 왼쪽 셀이 컴비네이터 M이면, 컴비네이터에 대한 감축규칙에 따라 그래프 감축을 수행한다. 컴비네이터 M은 먼저 자신의 인수 부그래프를 평가하여 그 결과로서 정수 n을 얻은 후, 배열을 생성하기 위한 기억장소로부터 n+1개의 노드를 할당받는다.

새로 할당받은 노드들은 (그림 4)의 b)와 같이 함수형 배열을 위한 자료구조를 형성한다. 원시배열에서 첫번째 노드는 배열 원소의 갯수를 나타내며, 각 노드의 왼쪽 셀은 갱신 정보를 나타내기 위한 nil로 초기화한다. 또한 감축 전의 노드1의 왼쪽 셀은 배열형 자료를 나타내는 형컴비네이터 ARR, 오른쪽 셀은 연결노드를 가리키는 포인터 값으로 갱신한다.



a) 감축 전의 그래프 형태 b) 감축 후의 그래프 형태

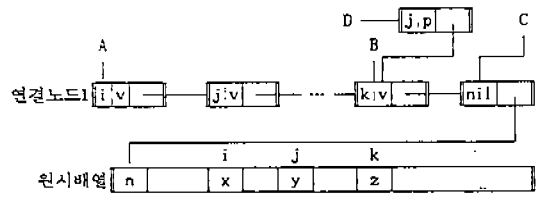
(그림 4) 컴비네이터 M의 감축규칙
(Fig. 4) Reduction rule for combinator M

(2) 갱신 연산자 upd(a, i, x) : (((D a) i) x)

함수형 배열에 자료를 갱신하기 위한 컴비네이터 D는 자신의 세번째와 두번째 인수 부그래프를 평가하여 각각 갱신할 자료 x와 인덱스값 i를 얻으며, 첫번째 인수 부그래프를 평가하여 배열에 대한 포인터 값을 얻는다.

최근 버전에 대한 갱신은 하나의 노드를 할당 받아서 원시배열의 내용을 저장하며 갱신될 내용을 직접 원시배열에 갱신한다. 그리고 과거 버전에 대한 갱신의 경우는 하나의 노드를 할당 받아서 해당되는 인덱스와 값을 저장한다. 이 노드는 새로 갱신된 버전의 첫번째 원소가 되며, 이 노드의 포인터 값은 연결 리스트의 정보 중에서 갱신될 버전이 가리키고 있는 원소를 지칭하게 한다. 갱신연산이 진행되는 과정에서 갱신된 원소에 대한 원시배열의 갱신 정보를 reset한다.

(그림 5)는 과거 버전 B에 대해서 j번째 원소를 p로 갱신한 후의 추상화된 그래프이다. 힙 기억장소로부터 하나의 노드를 할당받아서 인덱스 j와 값 p를 저장

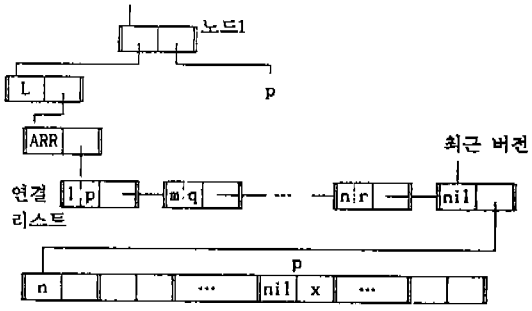


(그림 5) 과거 버전 B에 대한 갱신
(Fig. 5) Updating for old version B

하며 버전 B가 지칭하고 있는 연결 리스트의 포인터를 새 노드의 포인터 값으로 할당해 주며, 이와 같이 만들어진 노드가 새로운 버전 D의 시작노드가 된다. (그림 5)에서 최근 버전 C에 대해서 갱신이 이루어 진다면, 원시배열의 해당 원소가 새로 할당된 노드에 저장되며 이 노드가 버전 C의 초기노드가 되고, 원시배열에는 갱실될 내용이 직접 저장되며 새로운 최근 버전을 생성한다.

(3) 참조 연산자 sel(a, i) : ((L a) i)

배열로부터 자료를 참조하기 위한 연산인 컴비네이터 L은 두개의 인수를 취하여 감축을 수행한다. (그림 6)은 컴비네이터 L을 수행하기 전의 그래프 형태로서, 컴비네이터 L은 두번째 인수 부그래프를 평가하여 인덱스값 p를 얻으며, 첫번째 인수 부그래프를 평가하여 해당되는 배열의 포인터인 연결노드에 대한 포인터 값을 구한다.

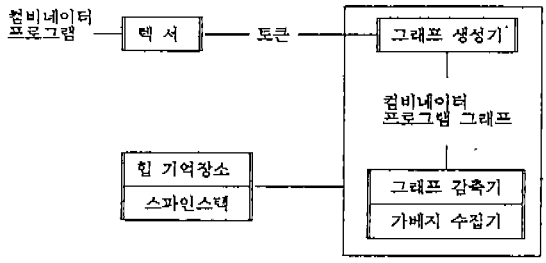


(그림 6) 컴비네이터 L에 대한 감축전의 그래프
(Fig. 6) Graph for combinator L, before reduction

이와 같이 컴비네이터 L이 필요로 하는 두개의 인수가 모두 평가되면 L에 대한 감축을 시작하는데 먼저 원시배열의 p번째 원소를 읽어서 갱신 정보가 nil이므로 x를 반환한다. 이는 p번째 원소가 갱신되지 않았음을 나타내므로 연결 리스트로부터 모든 노드를 탐색하지 않더라도 해당되는 값을 바로 반환할 수 있다. 그런데 원시배열의 갱신 정보가 nil이 아닐 경우는 해당되는 연결 리스트로부터 선형 탐색을 하여 값을 반환한다.

4. 함수형 배열의 구현

본 장에서는 함수형 배열의 처리를 위해서 3장에서 정의한 컴비네이터들을 구현하기 위한 컴비네이터 그래프 감축기의 구성과 감축 알고리즘을 나타낸다. 컴비네이터 그래프 감축 시스템은 렉서, 그래프 생성기, 그래프 감축기, 가베지 수집기로 구분할 수 있으며 구성은 (그림 7)과 같다.



(그림 7) 컴비네이터 그래프 감축기의 구성
(Fig. 7) Organization of combinator graph reduction

렉서는 함수 언어로부터 변환된 컴비네이터 프로그램을 입력으로 하여 토큰을 생성한 후 토큰을 그래프 생성기에 전달한다. 이 토큰들을 입력으로 하는 그래프 생성기는 컴비네이터 프로그램 그래프를 힙 기억장소에 형성하여 컴비네이터 프로그램 그래프의 루트 노드에 대한 포인터 값을 그래프 감축기에게 전달한다. 그래프 감축기는 스파인스택을 이용하여 수행될 컴비네이터를 루트 노드로부터 탐색하여 컴비네이터를 만나면 해당 컴비네이터의 감축규칙에 따라 실질적인 그래프 감축을 수행한다. 이 과정에서 필요로 하는 새로운 노드는 힙 기억장소로부터 할당받으며, 힙 기억장소 내에 더 이상 할당될 공간이 없으면 가베지 수집기가 수행된다.

[알고리즘 1]은 컴비네이터 그래프의 루트 노드에 대한 포인터 값을 입력으로 하여 수행 결과로 정수나 리스트, 또는 배열 등을 출력하는 컴비네이터 그래프 감축 알고리즘이다.

[알고리즘 1] Evaluator
(입력) combinator program graph
(출력) constant, list, array
(방법)
Evaluate(root_address of combinator program graph)

```
{
    Node_ptr←root_address;
      /* Node_ptr is a pointer variable for pointing
      of each node */
    while(1)
    {
      while (*(Node_ptr) is pointer) /* loop for
      searching of combinator */
    {
      PUSH Node_ptr;
      Node_ptr←*(Node_ptr);
    }
    switch(Node_Ptr) {
      case Turner Set Combinator:
        /* Execution of reduction according to
        each combinator */
      case Type combinator:
        /* BOOL, LIT, ARR */
      case Primitive operator:
        /* logical, relational, arithmetic opreators */
      case M:/* reduction rule of make_array
      operator */
        array_size←Evaluate(1st_arg);
        return_value←Construct_Array(array_size);
      case D:/* reduction rule of update operator */
        array←Evaluate(1st_arg);
        index←Evaluate(2nd_arg);
        int_value←Evaluate(3rd_arg);
        allocate new node;
        if (array is recent version)
            {updating original array;
              new node=(index, array[index])}
        else {new node=(index, int_value)
            pointing array}
        reset updating information;
        return_value←Array_updated();
      case L:/* reduction rule of select operator */
        array←Evaluate(1st_arg);
        index←Evaluate(2nd_arg);
        if (array[index]. left is nil) element←array
        [index].right;
        else element←Find_Element_Of_Array
        (array, index);
        return_value←element;
      default:
        Unknown combinator_error();
    }
  }
}
```

5. 실험 및 평가

본 논문에서 제안된 함수형 배열의 타당성을 검증하기 위하여 컴비네이터 그래프 감축기 상에서 다음과 같은 벤치마크 프로그램을 이용하여 HP-9000/700 계열의 워크스테이션 상에서 진행하였다.

- 벤치마크 1:배열의 초기화 및 모든 노드의 탐색: init_and_traverse()
- 벤치마크 2:배열을 이용한 피보나치 함수 프로그램:fibonacci()
- 벤치마크 3:배열의 sort 프로그램:array_sort()
- 벤치마크 4:연속적인 두 버전의 탐색:traverse_two_ver()
- 벤치마크 5:두 버전의 차를 구하는 프로그램:difference_two_ver()

벤치마크 1은 모든 배열의 원소를 임의의 값으로 초기화한 후에 이를 선형적으로 참조하는 프로그램이고, 벤치마크 2는 배열을 이용하여 피보나치 값을 구하는 것으로 인접한 값을 참조하는 프로그램이며, 벤치마크 3은 배열의 내용을 크기 순으로 나열하는 것으로 임의의 원소를 탐색하는 예로서 만들어진 프로그램이다. 이러한 벤치마크 프로그램들은 최근 버전을 주로 참조하고 갱신하는 경우의 예로서 본 논문의 효율성을 나타내기 위한 것이다. 벤치마크 4는 shallow 바인딩 방법과 비교하여 본 논문에서 발생될 수 있는 최악의 상태를 나타내기 위한 것으로 다른 두 버전을 연속적으로 참조하는 프로그램이다. 벤치마크 5는 shallow 바인딩 방법의 문제점을 나타내기 위한 것으로 두 버전에서 각각의 원소의 차를 구하기 위한 프로그램이다.

〈표 3〉 노드 탐색횟수의 비교
〈Table 3〉 Comparison of the node traversal

벤치마크 프로그램	트리형 배열	shallow binding	제안된 방법
init_and_traverse(100)	1400	400	300
fibonacci(30)	375	142	114
array_sort(20)	2380	1102	834
traverse_two_ver(20)	480	260	330
difference_two_ver(10)	200	195	115

〈표 3〉은 이와 같은 벤치마크 프로그램을 이용하여 그래프 감축이 진행되는 과정에서 배열을 처리할 때 발생되는 노드의 탐색횟수를 트리를 이용한 방법, shallow 바인딩 방법 등과 제안된 방법을 비교한 것이다. 각 벤치마크에서의 인자 값은 배열 원소의 갯수를 나타낸다.

트리를 이용한 방법은 매번 갱신을 위한 연산이 이루어질 때마다 $O(\log_2 n)$의 노드를 필요로 하며 배열의 임의의 원소를 검색하기 위한 시간도 $O(\log_2 n)$이므로, 배열 연산시 전체적인 노드 탐색횟수가 늘어난다. 또한 shallow 바인딩의 경우 다른 버전에 대한 참조가 발생될 때마다 환경을 재구성하기 위한 비용이 커지므로 노드의 탐색횟수가 증가한다.

본 논문에 의해 구축된 시스템에서는 최근 버전을 원시배열에 유지함으로써 최근 버전에 대한 참조가 O(1)에 가능하며, 다른 버전에 대한 참조를 위해 실행시간에 환경을 재구성하는 비용이 필요없으며, 과거 버전에 대한 판독이 이루어 질 경우 갱신 정보를 이용하여 탐색횟수를 줄일 수 있다. 벤치마크 4의 경우는 본 논문에서 구성한 함수형 배열에서 발생되는 최악의 경우로서 shallow 바인딩 방법 보다 탐색횟수가 증가되고 있다. 이는 shallow 바인딩이 같은 버전을 연속적으로 참조할 경우 갖게 되는 장점에 의한 것이지만, 벤치마크 5와 같이 자주 다른 버전을 참조하는 경우이거나, 최근 버전에 대한 참조가 빈번한 경우에서는 shallow 바인딩 방법 보다 효율적임을 알 수 있다.

6. 결론 및 향후과제

본 논문에서는 다중 쓰레드를 지원하는 함수형 배열에서 나타나는 기억공간 및 탐색시간 문제를 개선하기 위한 함수형 배열을 설계하고, 이를 컴비네이터 그래프 감축기 상에서 구현하였다.

제안된 함수형 배열은 갱신이 발생될 때마다 갱신되는 자료에 대한 추가적인 기억공간이 O(1) 공간이 되도록 하였으며, 갱신비용도 O(1)에 가능하도록 구성하였다. 본 논문은 배열을 이용한 많은 알고리즘이 최근 버전을 주로 참조한다는 관점에서 최근 버전을 원시배열에 유지하도록 하였으며, 다른 버전을 참조하는 경우 환경을 재구성하는 비용을 억제하였다. 이와 같이 최근 버전에 대한 탐색을 O(1)에 가능하도록 함으로써 명령형 언어가 갖는 배열의 특성에 근접할 수 있게 하였다. 제안된 모델의 타당성은 컴비네이터 그래프 감축기 상에서 벤치마크 프로그램을 이용하여 수행한 실험 결과를 통하여 탐색횟수가 감소됨을 보였다.

앞으로 추가적인 기억공간의 부담이 발생되더라도 연결 리스트 상에 존재하는 과거 버전에 대한 탐색시간을 최소화하기 위한 방법이 계속되어야 할 연구과제이다.

참 고 문 헌

[1] J. B. Backus, "Can Programming Be Liberated from the von Nuemann Style? A Functional Style and Its Algebra of Program", CACM, Vol. 21, No. 8, pp. 613-642, 1978.

[2] H. G. Baker, "Shallow Binding Makes Functional Arraya Fast", ACM SIGPLAN Notices, Vol. 26, No. 8, 1991.

[3] R. Bird and P. Wadler, Introduction to Functional Programming, Prentice Hall, 1988.

[4] A. Bloss, "Update Analysis and Efficient Implementation of Functional Aggregates", Functional Programming Languages and Computer Architecture, pp. 26-38, 1989.

[5] T-R. Chuang, "Fully Persistent Arrays for Efficient Incremental Updates and Voluminous Reads", 4th European Symposium on Programming, LNCS 582, pp. 110-129, Springer-Verlag, 1992.

[6] T-R. Chuang, "A Randomized Implementation

of Multiple Functional Arrays", ACM Conf. on Lisp and Functional Programming, pp. 173-184, 1994.

[7] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making Data Structures Persistent", Journal of Computer and System Sciences, Vol. 38, No. 2, pp. 86-124, 1989.

[8] K. Gopinath and J. Hennessy, "Copy Elimination in Functional Languages", 16th ACM Symposium on POPL, pp. 303-313, 1989.

[9] J. C. Guzman, "On Expressing the Mutation of State in a Functional Programming Language", YALE/DCS/RR-962, Yale Univ., 1993.

[10] J. C. Guzman and P. Hudak, "Single Threaded Polymorphic Lambda Calculus", Symposium on Logic in Computer Science, pp. 333-343, 1990.

[11] P. Hudak, "Arrays, Non-determinism, Side-effects, and Parallelism: a Functional Perspective", Worshop on Graph Reduction, LNCS 279, pp. 312-327, Springer-Verlag, 1986.

[12] P. Hudak, "Conception, Evoluation, and Application of Functional Programming Language", ACM Computing Surveys, Vol. 21, No. 3, pp. 359-411, 1989.

[13] S. L. Peyton Jones and P. Wadler, "Imperative functional programming", 20th ACM Symposium on POPL, pp. 71-83, 1993.

[14] S. D. Kim and W. H. Yoo, "New Combinators for the Efficient Execution of Functional Languages," Proc. InfoScience '93, pp. 381-385, 1993.

[15] P. J. Koopmann Jr., "A Fresh Look at Combinator Graph Reduction," ACM SIGPLAN Notices, Vol. 24, No. 7, pp. 110-119, 1989.

[16] M. Odersky, "How to Make Destructive Updates Less Destructive", 18th ACM Symposium on POPL, pp. 25-36, 1991.

[17] D. A. Turner, "A New Implementation Technique for Applicative Languages", Software Practices and Experience, Vol. 9, No. 1, pp. 31-49, 1979.

[18] P. Wadler, "A New Array Operation", Workshop on Graph Reduction, LNCS 295, Springer-Verlag, 1986.

[19] P. Wadler, "Comprehending monads", ACM Conf. on Lisp and Functional Programming, pp. 61-78, 1990.

[20] P. Wadler, "The essence of functional progrmming", 19th ACM Symposium on POPL, pp. 1-14, 1992.

주 형 석

1985년 인하대학교 전자계산학과 졸업(이학사)
1987년 인하대학교 대학원 전자계산학과 졸업(이학석사)
1994년 인하대학교 대학원 전자계산학과 박사과정 수료
1987년~현재 유한전문대학 전자계산과 부교수
관심분야: 프로그래밍 언어(함수언어, 객체지향언어), 계산모델, 병렬처리

유 원 희

1975년 서울대학교 공과대학 응용수학과 졸업(이학사)
1978년 서울대학교 대학원 계산학 전공(이학석사)
1985년 서울대학교 대학원 계산학 전공(이학박사)
1979년~현재 인하대학교 공과대학 전자계산공학과 교수
관심분야: 프로그래밍 언어(실시간 프로그래밍 언어, 함수 언어).