복잡한 자료 구조를 지니는 프로그램 슬라이싱

류 호 연[†] · 박 중 양[††] · 박 재 흥[†††]

요 약

프로그램 슬라이싱은 프로그램의 특정 지점에 나타난 변수의 값에 영향을 미치는 문장들을 프로그램으로부터 추출하는 방법이다. 프로그램 슬라이싱은 프로그램 디버깅, 프로그램 테스팅, 프로그램 통합, 병렬 프로그램 실행, 소프트웨어 메트릭스, 역공학, 유지보수 등 여러 응용 분야에 적용할 수 있다. 본 논문에서는 포인터 변수, 포인터 변수가 참조하는 객체, 배열, 구조체와 같은 복잡한 자료구조가 나타나는 프로그램에서 정확한 슬라이스를 추출하기 위한 알고리즘을 제안한다. 복잡한 자료구조가 나타나는 프로그램 상에서 객체의 보다 더 정확한 정적 분석 정보를 생성하기 위해 객체참조상태 그래프를 제안하고, 그 효율성을 보였다.

Program Slicing in the Presence of Complicated Data Structure

Ho-Yeon Ryu[†] · Joong-Yang Park[††] · Jae-Heung Park[†††]

ABSTRACT

Program slicing is a method to extract the statements from the program which have an influence on the value of a variable at a particular point of the program. Program slicing is applied for many applications, such as program debugging, program testing, program integration, parallel program execution, software metrics, reverse engineering, and software maintenance, etc. This paper is the study to create the exact slice in the presence of the complicated data structure including pointer variable, dereferenced object by pointer variable, array and structure. We propose the Object Reference State Graph to generate more exactly static analysis information of objects in the program of the presence of complicated data structure.

키워드 : 프로그램 슬라이싱(Program Slicing), 소프트웨어 유지보수(Software Maintenance), 소프트웨어 테스팅(Software Testing), 디버깅(Debugging)

1. 서 론

프로그램 슬라이싱은 프로그램의 특정 계산에 관련 있는 문장들을 추출하는 프로그램 분해 기법 중의 하나로, 프로그램 디버깅, 테스팅, 재사용 부품 추출, 프로그램 통합, 프로그램 최적화 등의 응용 분야에 적용할 수 있다. 프로그램 슬라이싱은 1984년, Mark Weiser[1]에 의해 처음으로 소개되어졌으며, 프로그램 실행 정보의 사용 유무에 따라 정적(static) 프로그램 슬라이싱과 동적(dynamic) 프로그램 슬라이싱으로 구분된다[6]. 정적 프로그램 슬라이싱은 프로그램 수행 전에 제어 흐름과 자료 흐름 정보를 기반으로 슬라이싱 기준에 영향을 주는 문장을 추출하는 방법으로, 전체 프로그램 코드의 이해를 지원하기 위해 유용하게 사용되어진다. 정적 프로그램 슬라이싱의 경우, 원시 프로그램을 분석함으로써 슬라이싱 기준에 영향을 줄 수 있는 모든 문장들을 추출한다. 따라서 특정 입력 값에 독립적인 슬라이스를 생성해 낼 수 있다. 동적 프로그램 슬라이싱은 프로그램 수행시, 특정 입력 값이 주어졌을 때 생성되는 프로그램 경로에 대하여 슬라이싱 기준에 영향을 주는 문장을 추출하는 방법으로써, Korel과 Laski[9]에 의해 최초로 제안되어 졌다. 동적 슬라이싱은 정적 슬라이싱보다 더 적은 슬라이스를 생성할 수 있으며, 오류가 발생한 초기 위치를 쉽게 파악할 수 있다. 또한 포인터 변수나 배열 그리고 동적 객체에 대한 분석이 가능하다. 동적 슬라이싱은 프로그램 실행시 부정확한 행위를 취하는 변수의 값에 영향을 주는 문장을 식별하는 디버깅에 유용하게 사용된다.

C 언어에서 처럼 포인터 변수, 포인터 변수가 참조하는 객체, 배열, 그리고 구조체와 같은 복잡한 자료 구조에 대한 정적 분석은 어렵다[3-5, 8]. 포인터 변수가 참조하게 되는 객체[1]에 대한 정확한 정보나 배열 첨자값이 변수로 나타날 경우 정확한 위치 정보, 그리고 동적 메모리 할당시

† 준 회 원 : 경상대학교 컴퓨터과학과 대학원
†† 정 회 원 : 경상대학교 통계정보학과 교수
††† 정 회 원 : 경상대학교 컴퓨터과학과 교수
논문접수 : 2003년 4월 8일, 심사완료 : 2003년 6월 16일

생성되는 객체의 수는 실행 시간에 결정되기 때문이다. 실제 프로그램 상에서 포인터나 배열, 구조체와 같은 복잡한 자료 구조를 발견할 수 있다. 배열 요소나 구조체의 필드는 접근 경로(access path)에 의해 참조될 수 있으며, 대부분의 포인터를 이용한 객체 참조시, 복잡한 연산식을 포함하지 않는다. 본 논문에서는 포인터 변수 연산식이나 배열 첨자값에 수식이 나타나는 경우를 허용한다.

본 논문은 포인터 변수와 포인터 변수에 의해 처리되는 객체, 배열, 그리고 구조체를 포함하는 프로그램에 대한 정확한 슬라이스 생성에 관한 연구이다. 복잡한 자료 구조를 지닌 객체들간의 참조 상태를 분석하고 표현하기 위하여 확장된 포인터 상태 부그래프(EPSS ; Extended Pointer State Subgraph), 재정의된 가상변수(Dummy Variable), 객체 참조 상태 그래프(ORSG ; Object Reference State Graph)를 이용한다.

본 논문의 구성은 다음과 같다. 2장에서는 복잡한 자료구조의 정적분석에 대한 기존의 연구들을 분석하고 3장에서는 복잡한 자료구조를 표현하기 위한 EPSS, 재정의된 가상변수, ORSG를 정의한다. 4장에서는 이를 프로그램 슬라이싱에 적용한 예를 보이고, 5장에서는 결론 및 향후 과제를 보이도록 한다.

2. 연구 배경

2.1 기존의 슬라이싱 방법

정적 프로그램 슬라이싱은 Weiser[1]에 의해 최초로 제안되었으며, 제안한 슬라이스의 정의는 다음과 같다.

정의 2.1 (슬라이스) 슬라이싱 기준 $c = \langle i, V \rangle$는 프로그램의 문장 번호 i과 프로그램에 나타나는 변수들의 부분 집합 V의 쌍으로 주어진다. 슬라이싱 기준이 주어졌을 때, 문장 i를 수행하기 직전까지 V에 있는 변수들에게 영향을 주었던 문장들을 추출하여 생성된 슬라이스는 동일한 입력값에 대해 원래 프로그램의 동작과 동일하며, 독립적으로 수행 가능한 코드이어야 한다.

Weiser의 정의에 따르면, 슬라이싱 기준 c가 주어졌을 때 i문장이 수행하기 직전까지 V에 있는 변수들에게 영향을 준 n(∈ Pred(i)) 문장에 나타난 변수들의 집합을 $RIN_c^0(n)$으로 나타낸다. 즉, $RIN_c^0(n)$에 있는 변수는 i노드 수행 직전까지 V에 있는 변수들에게 직접적으로 영향을 준 변수들의 집합이다. i노드 수행 직전까지 $RIN_c^0(n)$에 있는 변수들의 값이 변경되면, i노드 수행 후 V에 있는 변수들의 값은 원래 프로그램의 값과 달라지게 된다.

1) 변수 선언에 의해 정적으로 할당되거나, malloc 함수 호출에 의해 동적으로 할당되는 메모리 영역을 객체(object)라 한다.

슬라이싱 기준 $c = \langle i, V \rangle$가 주어졌을 때, $RIN_c^0(n)$을 구하는 공식은 다음과 같다.

$$RIN_c^0(n) = \{ v \in V \mid n = i \} \cup \{ Ref(n) \mid RIN_c^0(IMS(n)) \cap Def(n) \neq \emptyset \} \cup \{ RIN_c^0(IMS(n)) - Def(n) \}$$

$RIN_c^0(n)$ 정보를 이용하여 슬라이싱 기준에 나타난 V변수에 직접적으로 영향을 주는 문장을 추출하는 공식은 다음과 같다.

$$S_c^0 = \{ n \mid Def(n) \cap RIN_c^0(IMS(n)) \neq \emptyset \}$$

s_c^0에 어떤 제어문에 종속적인 문장이 포함되어졌다면, 그 문장에 대한 제어문도 추출해야하며, 제어문에서 참조된 변수에 대한 슬라이스도 포함해야 한다. 슬라이스에 포함된 문장에 대한 제어문을 추출하는 공식은 다음과 같다.

$$B_c^0 = \{ b \mid ND(b) \cap S_c^0 \neq \emptyset \}$$

첨자 0이 붙은 $RIN_c^0(n)$은 슬라이싱 기준에 나타난 V의 각 원소들에게 직접적인 영향을 주는 관련 있는 변수들의 집합이고, s_c^0는 직접적으로 영향을 주는 문장들의 집합이다. 슬라이스는 슬라이싱 기준에 나타난 변수들에게 직접적으로 영향을 주는 문장들뿐만 아니라, 간접적으로 영향을 주는 문장도 포함해야 한다. 따라서, 슬라이싱 기준 c에 직·간접적인 영향을 주는 변수들과 문장을 추출하기 위한 반복적인 방정식은 다음과 같다.

$$RIN_c^{i+1}(n) = RIN_c^i(n) \bigcup_{b \in B_c^i} RIN_{B_{c(b)}}^0(n)$$

$$S_c^{i+1} = \{ n \mid Def(n) \cap RIN_c^{i+1}(IMS(n)) \neq \emptyset \text{ or } n \in B_c^i \}$$

$$B_c^{i+1} = \{ b \mid ND(b) \cap S_c^{i+1} \neq \emptyset \}$$

$B_C(b)$은 제어문 b에 대한 슬라이싱 기준으로써 $\langle b, Ref(b) \rangle$이다. 슬라이싱 반복 수행은 다음의 조건을 만족하면 종료하게 된다.

$$\forall n \in N,\ RIN_c^{i+1}(n) = RIN_c^i(n) \text{ or } S_c^{i+1} = S_c^i$$

2.2 포인터 상태 부그래프

Lyle에 의해 제안된 포인터 상태 부그래프(PSS ; Pointer State Subgraph)는 포인터 변수가 사용된 프로그램에 대하여 포인터 참조 상태를 표현하기 위한 그래프이다[3]. 프로그램에서 포인터 변수가 가진 값(예를 들어, 객체의 주소값)을 계산하는데 필요한 문장들만을 추출하여 그에 대한 슬라이스를 구할 수 있게 된다.

PSS의 각 노드는 포인터 상태 함수(PSF ; Pointer State Function)로 설명 가능하며, 현재 위치 한 노드의 포인터 변수에 대한 포인터의 상태를 기술한다. 이 함수는 변수를 객체로 사상한다. 여기서 객체는 변수 선언에 의해 정적으로 또는 C 언어에서 처럼 malloc 함수 호출에 의한 동적으로 할당되는 저장소의 위치, NULL 등이 될 수 있다.

포인터 변수가 존재하는 문장에서만 포인터의 상태가 바뀌기 때문에 PSS는 전체 제어 흐름 그래프보다 훨씬 더 작다. 따라서, 모든 프로그램 문장들에 대해 이와 같은 상태를 결정하기 위해서 분석할 필요가 없다. 그러나, PSS에서는 포인터 변수에 대한 상태 변화에만 관심이 있기 때문에 그 외의 객체들에 대한 슬라이스는 구할 수는 없다.

2.3 저장소 형태 그래프

Livadas는 프로그램 실행시에, 힙 할당 저장소(Heap-allocated storage)가 존재하는 다양한 구조를 분석하기 위한 저장소 상태 그래프(SSG ; Storage Shape Graph)를 제안하였다[4]. Chase[5]에 의해 처음 제안된 이 그래프는 실행 시간(run-time) 동안에 할당되는 다양한 자료 구조에 대한 공간 효율적인 표현법을 포함한다. 즉, ANSI-C 언어에서 나타날 수 있는 복잡한 자료구조와 포인터 참조를 수용하고 자료 종속성 정보(data dependence information)을 알아내기 위해서 그래프 표기와 계산 알고리즘을 확장시켰다. 즉, SSG는 힙 할당 저장소에서 발생하는 앨리어싱과 동적 할당(dynamic allocation) 문제를 해결하기 위해 제안된 것이다.

SSG는 실행 후에 즉시 생성되는 것으로, 각 프로그램 지점 p에 대해 IN(p)와 OUT(p)의 두 단계를 거치게 된다. 여기서 IN(p)는 p가 실행되기 전에 존재하는 모든 메모리 패턴들의 집합을 나타내며, OUT(p)는 IN에서 나타났던 메모리에서의 문장들에서 변화가 발생한 것들을 나타내게 된다. 즉, SSG는 각 문장마다 두 개의 그래프의 쌍으로 이루어지는 반면, 본 논문에서 제안한 ORSG에서는 프로그램 당 한 개의 그래프만 생성된다.

3. 복잡한 자료구조를 지니는 프로그램

C 언어에서 처럼 포인터 변수, 포인터 변수에 의해 참조되는 객체, 배열, 그리고 구조체와 같은 복잡한 자료 구조에 대한 정적 분석은 어렵다[2-5, 8, 10]. 포인터 변수가 참조하게 되는 객체에 대한 정확한 정보나 배열 첨자값으로 변수가 나타나는 경우의 정확한 위치 정보, 그리고 동적 메모리 할당시 생성되는 객체의 수는 실행 시간에 결정되기 때문이다.

실제 프로그램 상에서 포인터, 배열, 구조체와 같은 복잡한 자료 구조의 사용을 발견할 수 있다. 일반적으로 스칼라 변수일 경우 그 변수를 하나의 객체로 취급하여 슬라이스 생성에 필요한 정확한 Def와 Ref 정보와 RIN 정보를 구할 수 있게 된다. 그러나 참조 연산이 발생하게 되는 배열 원소나 구조체 필드에 대해서는 부정확한 정보를 생성하게 된다.

본 절에서는 C 언어에서의 포인터 변수, 포인터 변수에 의해 참조되는 객체, 배열, 그리고 구조체와 같은 복잡한 정적 자료 구조를 지니는 프로그램에 대한 정확한 슬라이싱 방법을 기술하고, 기존의 방법보다 더 정확한 슬라이스를 생성할 수 있음을 보인다.

객체 참조 상태 그래프를 생성하기 위해 먼저 포인터 상태 부그래프가 생성되어져야 한다. 포인터 상태 부그래프는 제어 흐름 그래프에서 만들어진다.

3.1 확장된 포인터 상태 부그래프

프로그램 P에 대하여 포인터 상태 부그래프(PSS ; Point State Subgraph)는 CFG 상에서 포인터 변수의 값을 변경하는 노드를 식별함으로서 생성되어진다. 그러나 단순히 포인터 변수 상태만 변경하는 노드를 식별하는 것만으로는 배열 원소나 구조체 필드들에 대한 더 정확한 정보를 생성할 수 없다. 따라서, 본 논문에서는 기존의 PSS를 확장한 확장된 PSS(EPSS ; Extended Point State Subgraph)를 정의한다.

정의 3.1 (제어 흐름 그래프 ; CFG) : CFG는 프로그램 P의 문장들을 각각의 노드로 표현하는 방향성을 지니는 그래프(directed graph)로, 각 에지들은 프로그램 문장들간의 제어흐름을 표현하는 제어 흐름 에지(control flow edge)이다. Entry와 Exit는 프로그램의 진입점과 진출점을 나타내는 노드들이다.

정의 3.2 (포인터 상태 그래프 ; PSS) : PSS는 CFG의 부그래프로 포인터의 상태를 변경시키는 CFG의 노드들(예를 들어, p = &a, malloc 함수 호출)을 포함한다. PSS의 간선들은 제어 흐름 그래프로부터의 간선들의 경로이다.

정의 3.3 (확장된 포인터 상태 부그래프 ; EPSS) : EPSS는 CFG의 부그래프인 PSS를 확장시킨 것이다. 즉, EPSS는 CFG 상에서 포인터 변수에 값을 할당하는 노드 외에 배열 원소나 구조체 필드 접근 노드를 포함한다. EPSS는 CFG의 Entry 노드로부터 시작하여 CFG의 노드로부터 생성된다.

EPSS의 각 노드는 그 노드에서 나타나는 모든 포인터 변수들에 대해 포인터 상태 함수를 지니며, 이 함수는 포인터 변수가 참조하는 객체들의 집합을 함수값으로 반환한다.

정의 3.4 (포인터 상태 함수 ; PSF) : 노드 n에 나타나는 포인터 변수 v에 대하여, 포인터 상태 함수 P(n, v)는 v가

참조할 수 있는 객체의 집합이다. P(n, v)는 레벨단위로 분석되므로, $P_0(n, v)$는 v로부터 시작하여 0번 참조한 객체의 집합이 된다. $P_1(n, v)$의 경우, *v의 형태로 v가 1-레벨 참조할 수 있는 객체들의 집합을 의미한다.

P(n, v)는 다음과 같이 단계적으로 분해되어진다.

$P_0(n, v) = v$

$P_1(n, v) = \{ O | v$는 객체 o의 주소 $\}$ (O는객체 o의집합)

$P_k(n, v) = \{x \mid x \in P_1(n, y) \wedge y \in P_{k-1}(n, v)\}$

〈표 1〉 포인터 상태 전파 규칙($*^k$b는 b가 k번 참조되었음을 나타낸다)

Statement	Propagation Rule
a = &x	$P_1(n, a) = \{x\}$
a = b	$P_1(n, a) = P_1(n, b)$
a = *b	$P_1(n, a) = \cup P_1(n, x),\ \forall x \in P_1(n, b)$
a = **b	$P_1(n, a) = \cup P_1(n, x),\ \forall x \in P_2(n, b)$
a = $*^k$b	$P_1(n, a) = \cup P_1(n, x),\ \forall x \in P_k(n, b)$
*a = &x	$P_1(n, y) = P_1(n, y) \cup \{x\},\ \forall y \in P_1(n, a)$
$*^k$a = &x	$P_1(n, y) = P_1(n, y) \cup \{x\},\ \forall y \in P_k(n, a)$

포인터 상태를 변화시키는 문장에 대한 포인터 상태 함수 값을 계산할 수 있는 포인터 상태 전파 규칙(propagation rule)은 <표 1>과 같다. $P_k(k > 1)$의 값은 P_1의 값으로부터 EPSS의 각 제어 흐름 에지를 따라 전파 규칙에 의해 유도된다. 기존의 PSS의 수행 경로 상에 존재하는 노드의 포인터 상태 함수는 선행자 노드에서의 포인터 상태 함수 값을 포함한다.

① p = &a ;	P_1(①, p) = {a}
② p = &b ;	P_1(②, p) = {b}
③ *p = 5 ;	P_1(③, p) = {a, b}

(그림 1) 예제 프로그램과 PSF

예를 들어, (그림 1)은 포인터 변수가 나타나는 프로그램 일부분과 PSS로부터 생성되는 포인터 상태 함수 값을 보여준다. ②번 문장에서의 P_1(②, p) = {b}이고, ②번 문장 수행 후 포인터 상태 함수의 값은 동일한 포인터에 대한 함수값을 지니고 있는 선행자 ①의 값도 포함하게 된다. 따라서 P_1(②, p) = {b}가 된다. 순차적인 제어 흐름을 지니고 있는 경우, ③번 문장에서 포인터 변수 p의 상태 함수 값은 선행자인 ②번 노드의 포인터 상태 함수 값을 포함하여 P_1(③, p) = {a, b}가 된다. 그러나 프로그램을 자세히 살펴보면, 실제 ③번 문장에서 포인터 변수 p에 의해 참조되는 객체는 b이다. 즉, ③번 문장에서 포인터 변수 p는 더 이상 a 객체를 참조하지 않는다.

PSS는 포인터 변수에 대한 kill[2] 정보를 이용하지 않음으로서 부정확한 객체 참조 상태를 지니게 되어, 정확한 포인터 상태 함수값을 구하지 못한다. 그러나 EPSS는 포인터 변수에 대한 kill 정보를 유지함으로써 기존의 방법보다 더 정확한 객체 참조 상태 정보를 생성할 수 있다.

따라서, EPSS는 포인터 변수에 대한 kill 정보를 계산하기 위해 제어 흐름 에지 외에 하나의 에지가 추가되어야 하며 포인터 상태 함수도 변경되어야 한다.

정의 3.5 (정의-순서 에지 : Def-Order Edge[3]) : EPSS의 두 노드 n과 m은 다음의 조건을 만족하면 n은 m과 정의-순서 관계를 지니고 있다고 하고 $n \rightarrow_{do(p)} m$로 표현한다.

① 제어는 n에서 m으로 도달되어지며, 그들을 포함하는 조건문의 동일 가지(branch)에 존재한다.

② n과 m은 동일한 변수 p의 값을 정의한다.

③ n과 m을 제외한, n과 m 사이의 노드들은 p 값을 재정의하지 않는다.

정의-순서 에지가 추가된 EPSS에서 노드 m에 대한 포인터 상태 함수는 다음과 같이 변경되어진다.

$$[n \rightarrow_{do(p)} m] \rightarrow P_k(m, p) = P_k(m, p) - P_k(n, p)$$

프로그램 P에 대하여, EPSS는 다음의 규칙에 의해 생성되며 더 이상 적용시킬 경우가 없을 때까지 반복한다.

① 만일 노드 n이 단일 후속자(single successor) k와 단일 선행자(single predecessor) j를 가지고, 노드 n이 포인터의 상태를 변경시키지 않는다면, n을 제거하고 j와 k를 연결한다.

② 만일 노드 n이 여러 개의 후속자(multiple successor)를 갖고, 노드 j가 n을 포스트도미네이트(postdominate)하고, n과 j 사이에 있는 모든 노드들이 제거되면 그때 n과 j를 제거하고 n의 선행자와 j의 후속자를 연결한다.

③ 만일 노드 n과 m이 정의-순서 관계를 지니는 경우에 노드 n과 m을 정의-순서 에지로 연결한다.

④ Entry와 Exit 노드는 결코 제거되지 않는다.

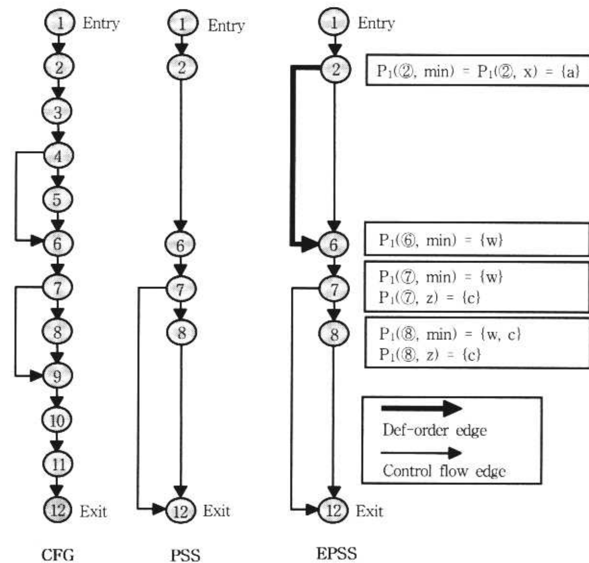
(그림 2)는 포인터 변수가 나타나는 함수와 이를 표현한 EPSS이다. Lyle의 방법을 이용한 PSS를 이용하여 각 노드에 대한 포이터 상태 함수값을 계산했을 때, P_1(②, min) = {a}, P_1(⑥, min) = {w}이 되므로 ⑥번 문장 수행 후 P_1(⑥, min) = {a, w} 값을 지니게 된다. 그러나 EPSS 생성 규칙을 적용한 경우, ②번과 ⑥번 노드가 ② $\rightarrow_{do(min)}$ ⑥ 관계를

2) 임의의 블록 외부에서 생성된 정의 중, 그 블록에서 재생성 되어지는 정의들의 집합.

3) PDG에서의 정의 순서 종속성 에지(Def-Order dependence Edge)와는 의미가 다르다.

갖게 되므로, 실제 P_1(⑥, min)의 값은 P_1(②, min) 값을 제거한 {w}가 된다. 따라서 EPSS를 이용한 포인터 상태 함수는 기존의 방법보다 더 정확한 포인터 상태 정보를 생성한다.

```
/*call : func (&a, &b, &c) return the square
 of the minimum of a, b, and c*/
            int func (int *x, int *y, int *z) {
①           int *min, square, w ;
②           min = x ;
③           w = *x ;
④           if (*y < w)
⑤            w = *y ;
⑥           min = &w ;
⑦           if (*z < *min)
⑧            min = z ;
⑨           w = *min ;
⑩           square = w*w ;
⑪           return square ;
⑫           }
```



(그림 2) 포인터 변수가 존재하는 예제 프로그램과 EPSS

3.2 객체 참조 상태 그래프

3.2.1 ORSG 정의

ORSG는 EPSS 노드 중 포인터 상태를 변경하는 노드의 포인터 상태 함수 값을 그래프로 표현한 것이다. ORSG를 이용하여 복잡한 자료 구조가 나타나는 프로그램에 대한 정확한 슬라이스를 생성할 수 있다. 그러나, 배열 원소나 구조체 필드 접근이 나타나는 노드에 대한 포인터 상태 함수 값은 정의되지 않기 때문에 ORSG를 이용하여 포인터 상태 함수를 계산할 수 없는 EPSS 상의 노드에 나타나는 객체들의 참조 상태를 표현한다. 실제 배열 인덱스 값으로 수식이 나타나는 경우 정확한 원소 위치 식별은 실행 시간이 결정된다. 따라서 배열 원소에 대한 정확한 정적 분석은 불가능하지만, 프로그램 문맥상 나타나는 배열 원소나 구조체 필드 접근 경로(access path)를 이용하여 기존의 방법들보다 더 정확한 정보를 생성할 수 있다.

본 절에서는 배열 원소나 구조체 필드 접근 연산이 발생하는 노드에 대하여 더 정확한 정보를 생성할 수 있는 ORSG를 정의한다.

정의 3.6 (객체 참조 상태 그래프 ; ORSG) : ORSG는 포인터 변수를 선언하는 문장과 포인터 변수에 값을 할당하여 상태를 변경시키는 문장, 그리고 배열 원소와 구조체 필드 연산이 발생하는 문장을 분석함으로써 생성된다. 즉, EPSS의 포인터 참조 상태와 배열 원소, 구조체 필드의 참조 상태를 표현한 그래프이다. 노드는 프로그램 상의 객체에 대응하고, 노드 n1과 n2간의 에지는 포인터(Point-to) 에지 관계와 멤버(member) 에지 관계를 나타낸다.

ORSG는 기본적으로 세 종류의 노드들을 지니며, 이들은 각각 세 종류의 부노드(subnode)로 분할되어질 수 있다. ORSG의 노드들은 저장소 형태 그래프(SSG ; Storage Shape Graph)의 노드 종류와 유사하다. ORSG는 변수(variable) 노드, 힙(Heap) 노드, 가상 힙(Virtual Heap) 노드로 이루어진다. 변수 노드는 이름 있는 메모리 영역을 나타내며 선언된 변수들이다. 예를 들어 C 언어에서의 기본 자료형(int, float, char)의 변수나 struct, union, enum, 그리고 이러한 자료형들에 대한 포인터 변수(int **a)가 포함된다. 힙 노드는 프로그램 실행시 동적으로 생성되어지는(malloc 함수 호출로 인한) 메모리 영역을 나타낸다. 예를 들어,

```
int *a ;
a = (long *)(malloc(sizeof(2 * char))) ;
```

의 경우 생성되어지는 힙 노드의 자료형은 ① 힙 영역을 참조하는 변수의 자료형, ② 캐스트 연산자에 의한 자료형, ③ sizeof 연산자에 의한 자료형이 될 수 있다. 그러나 본 논문에서 힙 영역에 할당되는 객체의 자료형은 그 객체를 가리키는 포인터 변수의 자료형과 동일하다고 가정한다.

가상 힙 노드는 배열의 구성 요소나 구조체의 필드처럼 정적으로 할당되지만 이름을 지니지 않는 메모리 영역을 나타낸다. 배열과 구조체 변수의 시작 부분은 이름이 명명되지만, 배열 구성 요소나 구조체의 필드들은 이들의 기본 주소(base address)로부터 오프셋(offset)값으로 처리되어진다. 따라서 배열 구성 요소와 구조체의 필드들은 가상 힙 노드로 표현되어진다. 세 가지 기본 노드들은 각각 스칼라, 포인터, 구조체 부노드들로 분할되어질 수 있다. <표 2>은 ORSG에서 9가지 노드 종류를 나타낸다.

ORSG는 두 종류의 에지를 지니고 있다. 첫째, 노드 n1에서 n2간의 포인터(Point-to) 에지 관계는 n1으로 표현된 포인터 변수나 힙 영역에 할당된 포인터 변수에 의해 노드 n2가 참조되어질 수 있음을 의미한다. 둘째 멤버(member)

에지는 프로그램 문맥상에 나타난 동일 배열이나 구조체에 대한 각 원소나 멤버들간의 접근을 나타내는 접근 경로들 간의 관계를 나타낸다.

모든 포인터 노드들은 자신과 관련된 간접 참조 레벨을 갖는다. 이는 포인터 변수 선언문에서 쉽게 결정된다. 예를 들어, C 언어에서 정수형 객체에 대한 포인터의 포인터 변수 x는 2-레벨 간접 참조를 지닌 포인터 변수이며, int **x 와 같이 선언된다.

〈표 2〉 ORSG에서의 노드의 9가지 종류

Node Name	Examples
Scalar Variables	int x ;
Point variable	float *x ;
Structure Variable	struct node1 x ;
Scalar Heap	x = (float *)(malloc(sizeof(float))) ;
Point Heap	x = (float **)(malloc(sizeof(float *))) ;
Structure Heap	x = (struct node1)(malloc(sizeof(struct node1))) ;
Scalar Virtual heap	struct node1 x ; (field X.i is Virtual node) int x[10] ; (body array x is Virtual Node)
Point Virtual heap	struct node2 x ; (field X.i is Virtual node) int *x[10] ; (body array x is Virtual Node)
Structure Virtual heap	struct node3 x ; (field X.i is Virtual node) struct node3 x[10] ; (body array x is Virtual Node)
struct node1 { int I ; } struct node2 { int *i ; }	struct node3 { struct node1 I ; }

3.2.2 ORSG 노드 스키마

3.2.2.1 변수 노드

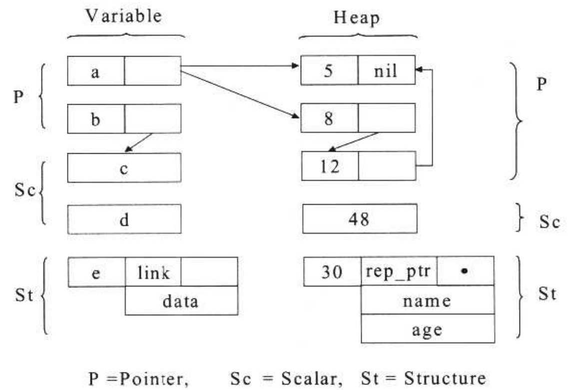
스칼라 변수에 대한 노드는 하나의 필드만을 지니며 자신의 이름이 레이블링 된다. 포인터 변수에 대한 노드는 두 개의 필드를 지닌다. 첫 번째 필드는 자신의 이름이 레이블링 되며, 두 번째 필드는 다른 노드로의 포인터 값이다. 포인터 변수인 경우에는 다중 참조(multiple reference)가 발생할 수 있으므로, 첫 번째 필드 위에 참조 레벨을 표기한다. 구조체 변수에 대한 노드는 구조체 변수 이름이 레이블링 되는 하나의 필드를 나타내는 노드와 구조체 자료형의 각 필드를 표현하는 부노드로 구성된다. 부노드는 노드들의 형태에 따라 스칼라, 포인터, 구조체 노드 표현을 따른다.

3.2.2.2 힙 노드

동적 메모리 할당 함수(예를 들어, malloc 함수)에 의해 프로그램 실행 시간에 생성되어지는 객체에 대한 것으로 힙 영역에 위치하게 된다.

동적 객체 할당 함수에 의해 힙 영역에 위치한 객체에 대한 스칼라, 포인터, 구조체 힙 노드들도 변수 노드들의 경우와 유사하다. 단 이름이 레이블링 되는 필드는 자신이 할당된 프로그램 상의 문번호를 레이블링 하게 된다. 즉, malloc 함수가 호출된 프로그램 상의 문장 번호가 힙 노드의 첫 번째 필드에 레이블링 된다.

변수 노드와 힙 노드에 대한 ORSG의 예는 (그림 3)과 같다. 다섯 개의 변수 노드 a, b, c, d, e에서 a, b는 포인터 변수이고 c, d는 스칼라 변수, e는 link와 data 필드를 지니는 구조체 변수이다 또한 다섯 개의 힙 노드가 있다.



(그림 3) 변수노드와 힙 노드가 존재하는 ORSG

3.2.2.3 가상 힙 노드

가상 힙 노드(Virtual Heap Nodes)는 정적으로 할당되어진 메모리 영역이지만, 프로그램에서 이름을 갖지 않는다. 예를 들어, 배열 객체 전체를 들 수 있다. 이는 이름을 가진 배열을 포인터가 참조하는 경우의 역을 생각하면 된다. struct 타입 변수의 필드도 마찬가지이다.

ORSG는 배열 원소나 구조체 필드들의 접근 상태를 표현할 수 있다. 한 문장에서 접근되는 배열 원소를 배열 객체(Array Object)라고 표현한다. 배열 객체들은 차수가 하나의 노드로 표현되는 링크드 리스트 구조로 표현된다. 각 배열 객체의 첫 번째 노드는 세 개의 필드를 지니는데, 첫 번째 필드는 배열 객체의 식별자가 레이블링된다. 배열 객체에 대한 식별자 규칙은 다음과 같다. 먼저 배열명을 중심으로 배열 객체가 문장 n에서 참조될 경우, 배열명 오른쪽에 문장 번호를 부착한다. 배정 연산자의 오른쪽에 여러 개의 동일 객체가 생성될 경우, (문번호-index, index $\geq$ 1)로 표현한다. 여러 문장에서 동일 배열 객체가 참조 될 경우는 배열명 오른쪽에 쉼표로 분리하여 문장 번호를 나열한다. 배열 객체가 정의될 경우는 배열명 왼쪽에 문장 번호를 부착한다. 배열 객체의 식별자는 배열 원소들 간의 자료 흐름 계산에 유용하다. 두 번째 필드는 배열 객체의 차수 노드가 표현되는 링크드 리스트로의 포인터이다. 차수 노드는 두 개의 필드로 나눠어 진다. 첫 번째 필드는 상수값이나 변수를 포함한 수식이 레이블링 되며, 두 번째 필드는 다음 차수 노드로의 포인터이다. 세 번째 필드는 다른 배열 객체에

대한 노드로의 포인터이다. 동일 배열 객체들은 링크드 리스트 구조로 연결되며, 이들의 첫 번째 노드는 배열명이 레이블링 되는 필드와 첫 번째 배열 객체 노드로의 포인터를 지닌다. (그림 4)는 3차원 배열을 포함하는 프로그램과 대응 ORSG이다.

두 개의 배열 객체 a1과 a2가 정확히 일치하거나 일치할 가능성이 있는 경우, 하나의 배열 객체로 합병되어질 수 있다. 다음은 두 배열 객체간의 정확한 일치(exact match)와 잠재적 일치(potential match)를 식별하는 방법이다.

두 배열 객체 ai와 aj는 다음의 조건을 만족하면 **정확히 일치(exact match)**한다고 한다.

① 어떤 객체도 수식을 포함해서는 안 된다.

② $N_{i,k} = N_{j,k}$, for $1 \le k \le$ len(ai), $N_{i,k}$는 배열 객체 ai의 k번째 노드의 레이블이다. len(ai)는 배열 객체의 첫 번째 노드를 제외한 차수 노드의 길이이다.

두 배열 객체 ai와 aj는 다음의 조건을 만족하면 **잠재적으로 일치(potential match)**한다고 한다.

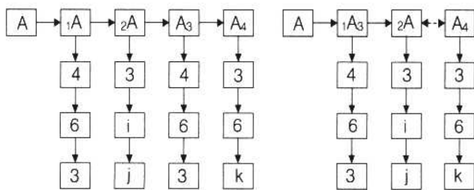
for $1 \le k \le$ len(ai),

① $N_{i,k}$나 $N_{j,k}$ 둘 중 하나는 수식이거나,

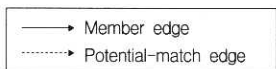
② $N_{i,k} = N_{j,k}$

```
        int x, y, z ;
        int A[10][10][10] ;
        int I, j, k ;
        sub() {
①           A[4][6][3] = x ;
②           A[3][i][j] = x ;
③           b = A[4][6][3] ;
④           c = A[3][6][4] ; }
```

(a)



(b) (c)



(그림 4) 다차원 배열과 ORSG

배열 객체 노드들이 정확히 일치 관계를 지니고 있다면, 두 노드는 합병되어질 수 있다. 잠재적 일치 관계를 지니고 있다면, ORSG 상의 배열 객체간에 잠재적 에지(potential edge)를 연결한다. (그림 4)(b)에 대하여 $_1$A와 A_3 배열 객체는 정확히 일치하고, $_2$A와 A_4는 잠재적으로 일치한다. 그러나 $_1$A와 $_2$A는 일치 관계가 없다. 따라서 $_1$A와 A_3 배열 객체는 $_1A_3$의 이름으로 합병되어지며, $_2$A와 A_4는 잠재적 에지 관계를 지니게 된다. (그림 4)(b)에 대하여 합병되어진 배열 객체의 ORSG는 (그림 4)(c)와 같다.

구조체 필드들에 대한 접근은 배열 객체의 접근법과 유사하다. 구조체는 중첩된 구조로 나타날 수 있으며, 각각의 구조체에 대한 접근 경로들을 구조체 객체(Structure Object)라고 하며, 구조체 객체의 각 노드는 필드 식별자로 레이블링 된다.

두 구조체 객체 ai와 aj는 다음의 조건을 만족할 경우 **정확히 일치**한다.

for $1 \le k \le minlen(ai, aj)$, ($minlen$ 함수 : ai와 aj중 더 적은 노드 개수를 반환).

① 어떤 객체도 수식을 포함하지 않는다.

② $N_{i,k} = N_{j,k}$, $N_{i,k}$는 구조체 객체 ai의 k번째 노드의 식별자 대응 필드 노드의 식별자가 모두 동일할 경우 두 객체는 정확히 일치한다.

두 구조체 객체 ai과 aj는 다음의 조건을 만족할 경우 **잠재적 일치**한다.

① $N_{i,k}$나 $N_{j,k}$ 둘 중 하나는 수식이거나,

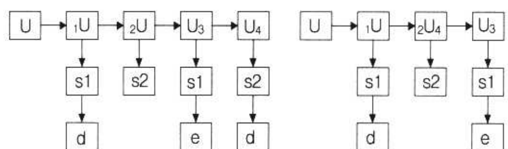
② $N_{i,k} = N_{j,k}$,

(그림 5)는 구조체를 지닌 프로그램과 대응 ORSG이다. $_2$U와 U_4는 정확히 일치하며 $_1$U와 U_3은 일치 관계가 없다.

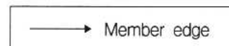
```
    int a, b ;
        typedef struct {
                int d, e ;
        } stype1 ;
        struct {
                stype s1, s2 ; }
        } U ;
        stype1 V ;

        sub2() {
①               U.s1.d = 10 ;
②               U.s2 = V ;
③               a = U.s1.e ;
④               b = U.s2.d ; }
```

(a)



(b) (c)



(그림 5) 구조체가 존재하는 예제 프로그램과 ORSG

슬라이싱 기준에 나타난 객체가 다른 객체와 잠재적 에지 관계를 지니고 있다면, 그 객체에 대한 슬라이스도 함께 포함해야한다. 본 논문에서 제안하는 슬라이싱 방법에 대한 시간 복잡도는 선형시간으로 $O(n)$과 같다.

3.3 가상 변수(Dummy Variable)

포인터와 포인터 변수에 의해 접근 가능한 메모리 영역을 다루기 위해 각 포인터 변수에 대한 가상 변수를 사용한다[2]. ANSI-C에서 **p 형식의 포인터 변수 p에 대한 가상 변수는 (1)p와 (2)p이다. 포인터 변수 앞에 붙은 상수값은 포인터 변수 p의 간접 참조 레벨 수를 의미한다. 이때 p = (0)p이다. 포인터 변수에 값을 할당하는 p = &x 문장의 경우, 주소 연산자가 붙은 x 변수에 대한 가상 문자(dummy literal)는 (-1)x로 표현한다.

다음 프로그램 일부분을 고려해 보자.

```
1. p = &a ;      Def(1) = {p},        Ref(1) = {}
2. *p = 3 ;      Def(2) = {(1)p},     Ref(2) = {p}
3. *p = 5 ;      Def(3) = {(1)p},     Ref(3) = {p}
4. print(a) ;    Def(4) = {},         Ref(4) = {a}
```

슬라이싱 기준 c = <4, a>로 주어졌을 때, 앞서 보인, Jingyue의 방식으로 계산된 슬라이스는 1, 2, 3, 4번 문장을 포함하게 된다. 그러나 프로그램을 자세히 살펴보면, 2번 문장에서 할당된 값은 3번 문장에 의해 kill 되어진다. 따라서 슬라이스에 포함되는 문장은 1, 3, 4번 문장이다.

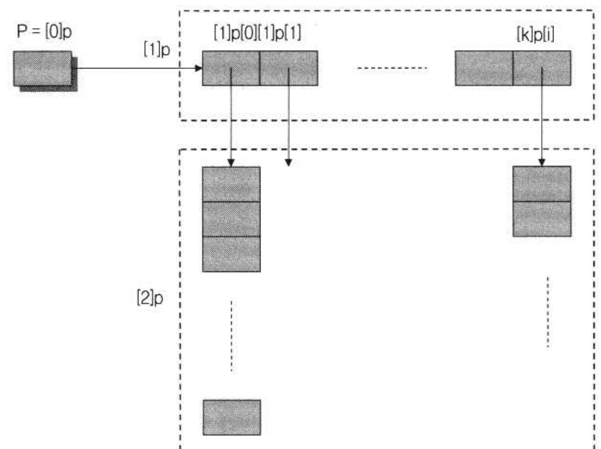
〈표 3〉 Jingyue 방법을 사용한 예제

Statement		Def	Ref	RIN
1	*(p+k) = c1	(1)p	p, k, c1	p, i, j, k
2	*(p+i) = c2 ;	(1)p	p, i, c2	p, i, j
3	*(p+j) = c3 ;	(1)p	p, j, c3	p, i, j, (1)p
4	if (e)		e	p, i, j, (1)p
5	k = i ;	k	i	p, i, (1)p
6	else			
7	k = j ;	k	j	p, j, (1)p
8	x = *(p+k) ;	x	p, k, (1)p	p, k, (1)p
9	printf(x)		x	

<표 3>은 포인터 연산식이 나타나는 프로그램과 슬라이싱에 필요한 분석 정보이다. 슬라이싱 기준이 <9, x>로 주어졌을 때, Weiser[1]의 알고리즘을 적용하여 생성된 슬라이스는 {3, 4, 5, 7, 8, 9}이다. 그러나 프로그램을 자세히 살펴보면, 8번 문장에 나타난 포인터 변수 p에 대한 연산식에서 정확히 p가 참조하는 객체에 대한 정보를 얻을 수가 없다. Jingyue[2]의 방식에 의하면 포인터 변수 p에 대한 연산식이 나타나는 1과 2번 문장도 슬라이스에 포함한다. 그러나 프로그램을 더 자세히 살펴보면 1번 문장과 8번 문장에서 나타나는 포인터 연산식 (p+k)는 5번과 6번 문장에서 k의 값이 재정의됨으로 인해 실제 동일 객체를 참조하지 않는다. 따라서 1번 문장은 슬라이스에 포함될 필요가 없다.

포인터나 배열, 구조체가 존재하고 포인터 연산식과 배열 첨자값으로 수식이 나타나는 경우, 기존의 연구들은 부정확한 슬라이스를 생성하게 된다. 실제, 포인터 변수에 대한 연산은 포인터 변수가 배열을 가리킬 때 나타난다. 즉 배열에 대한 연산은 배열 각각의 모든 원소에 대해서 동일하게 적용되기 때문에 포인터 연산식은 하나의 프로그램 내에서 유일하게 나타나는 경우가 대부분이다. 따라서 본 논문에서는 배열이나 구조체 링크드 리스트 자료 구조를 가리키는 포인터 변수에 대한 연산식은 하나의 프로그램 내에서 유일하다고 가정하고, 가상 변수를 정의한다.

본 논문에서는 Jingyue에서의 방식과는 조금 다른 포인터 변수와 배열의 가상 변수 표현법을 사용한다. k-레벨 간접 참조하는 포인터 변수 p에 대하여, [k]p[i]로 나타낸다(k ≥ 0, i는 0 혹은 변수명 집합). k값을 좌첨자라 하고, i값을 우첨자라 한다. k는 간접 참조 레벨수를 나타내기 위한 음이 아닌 정수값이다. i는 포인터 변수 연산식에 나타난 변수들을 원소로 지니는 집합이거나, 연산식이 나타나지 않을 경우 0(zero)을 값으로 지닌다. 예를 들어, 포인터 변수 p에 대한 *(p+i)과 같은 포인터 변수 연산식이 나타나는 경우 p에 대한 가상 변수는 [1]p[i]로 나타낸다. *p의 경우는 [1]p[0]이고 좌·우첨자가 0(zero)인 경우는 생략 가능하다. 즉 *p = [1]p로 표현한다. (그림 6)은 가상 변수를 나타낸 것이다.



(그림 6) 새로운 가상 변수 예제

<표 4>는 새로운 가상변수 표현법을 적용한 경우이다. 첫 번째 열은 원시 프로그램 문장들이고, 두 번째와 세 번째 열은 각 문장에서의 계산된 분석 정보이다. 네 번째 열

은 슬라이싱 기준 c = <8, x>로 주어졌을 때 계산된 RIN 정보이다.

<표 4>를 자세히 살펴보면, RIN(7)에 포함된 가상 변수 [1]p[k]가 RIN(6)에서는 [1]p[j]로 바뀌었다. 즉, 6번 문장에서 k의 값이 j의 값에 직접적인 영향을 받기 때문에 가상 변수의 우첨자 리스트에 나타나는 k 이름도 j로 바뀐다. 따라서 [1]p[k]는 [1]p[j]로 재명명 되어진다.

〈표 4〉 새로운 가상변수를 적용한 예제

Statement		Def	Ref	RIN
1	*(p+k) = c1	[1]p[k]	p, k, c1	p, i, j
2	*(p+i) = c2 ;	[1]p[i]	p, i, c2	p, i, j
3	*(p+j) = c3 ;	[1]p[j]	p, j, c3	p, i, j, [1]p[i]
4	if (e)		e	p, i, j, [1]p[i]. [1]p[j]
5	k = i ;	k	i	p, i, [1]p[i]
6	else			
7	k = j ;	k	j	p, j, [1]p[j]
8	x = *(p+k) ;	x	p, k, [1]p[k]	p, k, [1]p[k]
9	printf(x)		x	

먼저 문자를 원소로 지니는 집합 S에서 원래 원소 a를 원소 b로 교체하여 새로운 집합 S′을 생성하는 Rep 연산을 다음과 같이 표기한다.

$$a \in S, \quad S' = Rep(S)_{a \to b}$$

이 기호로서 가상 변수의 재명명에 대한 공식을 정의하면 다음과 같다.

슬라이싱 기준 c가 주어졌을 때, m ∈ IMP(n)에 대하여,

$$[k]p[S] \in RIN(n) \ \wedge \ \exists a \in S,\ a \in Def(m) \Rightarrow$$
$$[k]p[S'] \in RIN(m),\ S' = Rep(S)_{a \to Ref(m)}$$

노드 n에 나타난 가상 변수 [1]p는 P1(n, p)의 값을 지닌다. 따라서 RIN의 값에 가상 변수가 나타나면 EPSS 상에서 계산된 포인터 상태 함수 값으로 치환 되어진다.

4. 프로그램 슬라이싱

프로그램 P에 대해서, 특정 슬라이스 기준에 의해서 추출된 슬라이스에 포함된 문장이 포인터에 의한 간접 참조나 배열, 구조체에 대한 연산을 포함한다면, 포인터 변수와 그 포인터 변수에 의해 참조될 수 있는 모든 변수들, 배열, 구조체 객체에 대한 슬라이스도 계산해야한다.

먼저, 포인터 변수가 1-레벨 간접 참조(indirect reference)인 경우를 살펴보자.

$$n : \cdots = \cdots *p \cdots ;$$

의 경우, 문장 n이 슬라이스에 포함된 경우, 포인터 변수 p와 p의 포인터 상태 함수 값에 있는 모든 변수들에 대한 슬라이싱도 수행되어야 한다. 따라서 필요한 슬라이스는 다음과 같다.

$$S_{\langle n, P_0(n,p)\rangle} \cup (S_{\langle n, P_1(n,x)\rangle}\ where\ (x,1) \in irefs(n)).$$

예를 들어, 만일 포인터 변수 p에 대한 1-레벨 포인터 상태 함수값이 $P_1(n, p)$ = {E, F}라면, *p의 값에 의해 참조되어지는 모든 경우를 고려하여 구하게 되는 새로운 슬라이스는 $S_{<n, E>}$, $S_{<n, F>}$, 그리고 $S_{<n, p>}$가 된다.

k-레벨(k ≥ 1) 간접 참조인 경우를 일반화하는 경우는 다음과 같다.

$$n : \cdots *^k p \cdots$$

만일 문장 n이 슬라이스에 포함되면, 그때 다음의 슬라이스들 또한 슬라이싱 되어야만 한다.

$$S_{\langle n, P_i(n,p)\rangle}\ where\ (x,k) \in irefs(n)\ \text{and}\ 0 \le i \le k.$$

또한, 슬라이스에 포함되지 않은 문장이라도 해당 포인터 변수의 포인터 상태 함수 값에 슬라이싱 기준에 나타난 변수가 포함된다면 또한 그 포인터 변수에 대한 슬라이싱도 수행해야 한다. 이는 다중 레벨의 간접 참조를 가지므로 복잡하다. 다음을 경우를 고려해 보자.

$$n : *p = \cdots ;$$

슬라이스 기준 c = <succ(n), c>에 대한 슬라이스를 구하고자 할 때, $P_1(n, p)$가 c를 포함하면 슬라이스에 문장 n을 포함한다. 문장 n에서 p의 값에 영향을 미치는 문장들에 의해서 슬라이스는 생성된다. 예를 들어, 만일 문장 p = &c가 p의 값을 변경시켰다고 한다면, 슬라이스 $S_{<n, p>}$로 p의 값이 변경됨을 알 수 있다. 만일 p가 또 다른 변수를 참조한다면 그때 슬라이스 $S_{<n, c>}$도 c에 다른 값이 할당되어짐을 알아내는데 사용된다. k-레벨(k≥1) 참조인 다음의 경우를 살펴보자.

$$n : *^k p = \cdots ;$$

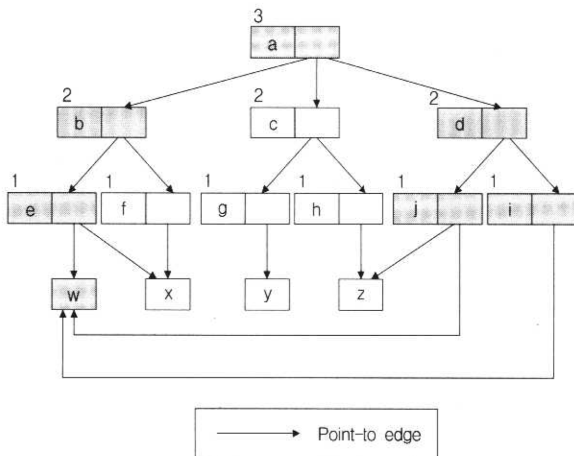
문장 n에서 v가 $*^k p$가 참조하는 어떤 변수라고 한다면, n은 슬라이스 $S_{<succ(n),v>}$에 포함된다. 예를 들어, v ∈ $P_k(n, p)$이고 (p, k) ∈ idefs(n)이라면 슬라이스에 문장 n이 포함된다. 따라서, k-레벨(k ≥ 1) 참조하는 포인터 변수의 경우 ORSG 상에서 직접적으로 들어오는 포인터 에지 관계를 지닌 중간 포인터 변수에 대한 슬라이싱을 수행해야한다. 즉, 생성된 추가적인 슬라이스들은 오직 관련 있는 중간 단계의 포인터 참조에 대한 슬라이스까지 포함해야만 한다. 이

를 위해서는 관련 중간 참조 함수(relevant intermediate indirect references function) $R_{i,k}(n, v, x)$가 필요하다. 이 함수 R은 변수 x에서 변수 v로 k 레벨의 간접 참조를 사용함으로써 문장 n에서 할당을 위해 레벨 i 간접 참조에 사용될 수 있는 중간 포인터의 집합을 반환하여, 효과적으로 특정 슬라이싱 기준과 관련이 없는 포인터 참조를 제거할 수 있게 한다. $P_i(n, v)$은 v가 i-레벨 참조하는 변수들의 집합이지만, 실제로는 (k-i)레벨 참조 후, x를 참조하는 $P_i(n, v)$의 집합이 슬라이싱에 관련이 있다.

$$R_{i,k}(n,v,x) = \{r \mid r \in P_i(n,x) \text{ and } v \in P_{k-i}(n,r)\}$$

(그림 7)은 포인터가 나타난 프로그램 일부분과 그에 대한 ORSG를 표현한 것이다. 여기에서는 포인터 변수 a에 세 번에 걸친 다중 참조가 발생한다.

```
{
  int w, x, y, z, *e, *f, *g, *h, *i, *j, **b, **c, **d, ***a ;
   :
   a = cond( )? ( cond( )? &b : &c ) : &d ;
   :
   b = cond( )? &e : &f ;
   c = cond( )? &g : &h ;
   d = cond( )? &i : &j ;
   :
   e = cond( )? &w : &x ;
   f = &x ; g = &y ; h = &z ; i = &w ;
   j = cond( )? &w : &z ;
   :
  n : ***a = …
   :
}
```



(그림 7) 포인터 변수가 존재하는 프로그램과 ORSG

슬라이싱 기준이 $S_{<succ(n), w>}$로 주어질 때, 문장 n이 슬라이스에 포함되면 3-레벨 참조를 지니는 포인터 변수 a의 값에 영향을 미치는 중간 단계의 포인터 변수, b, d, e, i, j에 대한 슬라이스도 포함되어야만 한다. 즉, R(n, w, a)를 포함시킨다.

따라서, 만일 $S_{<succ(n), v>}$가 문장 n을 포함한다면 추가로, 다음 슬라이스들을 포함해야 한다.

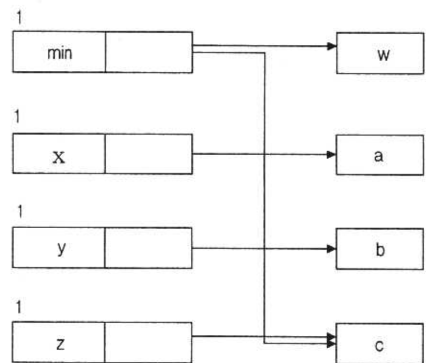
$$S_{\langle n, R_{i,k}(n,a,v)\rangle} \ where\ 0 < i < k \text{ and } (a, k) \in idefs(n)$$

또한, 배열 객체나 구조체 객체가 존재하는 프로그램의 경우도 프로그램 문맥상의 접근 경로를 ORSG에서의 재명명된 객체 이름으로 변경하는 작업을 수행 한 후, 슬라이싱을 수행한다. 단, ORSG 상에서 잠재적 에지 관계를 지닌 객체에 대한 슬라이싱도 포함해야 한다.

포인터 변수가 존재하는 (그림 8)에서 슬라이싱 기준 c = <⑪, square>로 주어졌을 때 슬라이스 생성 과정을 살펴보면 다음과 같다. 먼저 생성된 EPSS로부터 ORSG를 생성해야한다. EPSS의 각 노드로부터 계산된 포인터 상태 함수는 <표 5>과 같다. ⑧번 문장 이후로는 포인터 상태를 바꾸는 문장은 존재하지 않으므로 ⑧번 문장 이후부터 'Exit'노드까지의 ORSG는 (그림 8)과 같다. 그리고 각 문장에 대한 분석 정보는 <표 6>과 같다.

〈표 5〉 EPSS에서의 PSS

P_1('Entry', x) = {a}, P_1('Entry', y) = {b}, P_1('Entry', z) = {c}
P_1(②, min) = P_1(②, x) = {a}
P_1(⑥, min) = {w}
P_1(⑦, min) = {w}, P_1(⑦, z) = {c}
P_1(⑧, min) = {w, c}, P_1(⑧, z) = {c}



(그림 8) ORSG

기존의 방법으로 생성된 슬라이스는 문장 전체 포함한다. 그러나 프로그램을 자세히 살펴보면 ②번 노드에서의 min 변수는 square 변수의 값을 변경시키지 않는다. 그러나 Def, Ref 정보와 계산된 RIN 정보를 이용하여 자료 흐름 방정식을 이용하여 생성된 슬라이스는 ③, ④, ⑤, ⑥, ⑦, ⑧, ⑨, ⑩, ⑪번 노드이다. 따라서 본 연구의 가상 변수를 이용한 복잡한 자료 구조를 지닌 프로그램에 대한 슬라이싱은 기존의 방법보다 더 정확한 슬라이스를 생성함을 알 수 있다.

〈표 6〉 분석정보

노드 번호	Def	Ref	RIN
①	x, y, z	[-1]a, [-1]b, [-1]c	P_1(②, z) = P_1(②, min) = {c} P_1(②, y) = {b}, P_1(②, x) = {a}
②	min	x	z, x, y, P_1(②, z) = P_1(②, min) = {c}, P_1(②, y) = {b}, P_1(②, x) = {a}
③	w	[1]x	z, x, y, P_1(③, z) = P_1(③, min) = {c}, P_1(③, y) = {b}, P_1(③, x) = {a}
④		[1]y w	z, y, P_1(④, z) = {c} P_1(④, min) = {w, c}, P_1(④, y) = {b}
⑤	w	[1]y	z, y, P_1(⑤, z) = {c}, P_1(⑤, min) = {c}, P_1(⑤, y) = {b}
⑥	min	[-1]w	z, P_1(⑥, z) = {c}, P_1(⑥, min) = {w,c}
⑦		[1]z [1]min	min, z, P_1(⑦, z) = {c} P_1(⑦, min) = {w, c}
⑧	min	z	z, P_1(⑧, z) = {c}, P_1(⑧, min) = {w, c}
⑨	w	[1]min	min, P_1(⑨, min) = {w, c}
⑩	square	w	w
⑪		square	square

5. 결론 및 향후 과제

프로그램 슬라이싱은 특정 계산에 관련 있는 문장 추출에 기반을 둔 프로그램 분해 방법으로, 프로그램 디버깅, 최적화, 프로그램 유지 보수, 테스팅, 재사용 부품 추출, 그리고 프로그램 이해를 포함하는 여러 응용 분야들에서 그 유용성을 확인할 수 있다. 본 논문은 포인터 변수, 포인터 변수가 참조하는 객체, 배열, 그리고 구조체와 같은 복잡한 자료 구조를 지니는 프로그램에 대한 정확한 슬라이스 생성에 관한 연구이다. 포인터 연산식과 배열 원소, 구조체 필드 접근 연산이 존재하는 경우 확장된 포인터 상태 부그래프, 개선된 가상변수 표현법, 그리고 객체 참조 상태 그래프를 이용하게 되면, 기존의 방법보다 더 정확한 정적 분석 정보를 생성함으로써 더 정확한 슬라이스를 생성할 수 있다.

향후 과제로는 동적 객체의 무제한적 생성시에 발생하게 되는 슬라이싱 문제를 해결하는 것과 자동화된 슬라이서 도구 개발이다.

참 고 문 헌

[1] Mark Weiser, "Program Slicing," IEEE Trans. Software Eng., Vol.SE-10, No.4, pp.352-357, July, 1984.

[2] Jingyue Jiang, Xiling Zhou, David Robson, "Program Slicing For C-The Problems in Implementation," Proc. IEEE International Conf. Software Maintenance, pp.182-190, 1991.

[3] J. R. Lyle and D. W. Binkley, "Application of the Pointer State Subgraph to Static Program Slicing," to appear in the *Journal of Systems and Software* during the fourth quarter of 1997 or the first quarter of 1998.

[4] Pandos E. Livadas, Adam Rosenstein, "Slicing in the Presence of Pointer Variables," Ghinsu Project Technical report, 1995.

[5] D. Chase, M. Wegman and F. Zadeck, "Analysis of Pointers and Structures," *Preceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, New York pp.296-309, June, 1990.

[6] S. Horwitz, T. Reps and D. Binkley, "Interprocedural slicing using dependence graphs," ACM Transactions on Programming Languages and systems, 12(1), pp.35-46, January, 1990.

[7] K. B. Callagher and J. R. Lyle, "Using program slicing in software maintence," *IEEE Transactions on Software Engineering*, 17(8), pp.751-761, August, 1991.

[8] S. Horwitz, P. Pfeiffer and T. Reps, "Dependence analysis for pointer variables," *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, (Portland, OR, June, 1989), ACM SIGPLAN Notices 24(7), pp.28-40, July, 1989.

[9] B. Korel and J. Laski, "Dynamic Program Slicing," *Information processing letters*, Vol.29, No.3, Oct., 1988.

[10] Agrawal, H., Demillo, R. and Spafford, E., "Dynamic slicing in the presence of unconstrained pointers," *In Proceedings of the ACM Proth Symposium on Testing, Analysis and Verification*, pp.60-73, 1991.

[11] W. E Weihl, "Interprocedural Data Flow Analysis in the Presence of Pointers, procedure Variables and Label Variables," *In Conference record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pp.83-94, January, 1980.

[12] W. Landi and B. Ryder, "Safe Approximate Algorithm for Interprocedural Pointer Aliasing," *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp.235-248, June, 1992.

[13] M. Weiser, "Programmers Use Slices When Debugging," *CACM 25(7)*, pp.446-452, July, 1982.

[14] James R. Lyle, David Binkley, "Program Slicing in the Presence of Pointers," *Preceedings of the 3RD Annual Software Engineering Research Forum*, November, 1993.

류 호 연

e-mail : nollai@magicn.com

1994년 ~ 1998년 경상대학교 컴퓨터과학과 (학사)

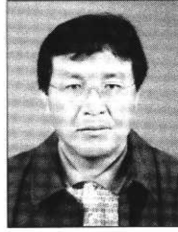
1998년 ~ 2000년 경상대학교 컴퓨터과학과 (석사)

2000년 ~ 현재 경상대학교 컴퓨터과학과 (박사과정)

관심분야 : 소프트웨어 공학, 테스팅, 소프트웨어 에이전트, 보안, 정보전

박 중 양

e-mail : parkjy@nongae.gsnu.ac.kr
1982년 연세대학교 상경대학 응용통계학과 (학사)
1984년 한국과학기술원 산업공학과 응용 통계전공(석사)
1994년 한국과학기술원 산업공학과 응용 통계전공(박사)
1984년~1989년 경상대학교 전산통계학과 교수
1989년~현재 경상대학교 통계정보학과 교수
관심분야 : 소프트웨어 신뢰성, 신경망, 선형통계 모형, 실험 계획법 등



박 재 흥

e-mail : pjh@nongae.gsnu.ac.kr
1973년~1978년 충북대학교 수학과(학사)
1978년~1980년 중앙대학교 대학원 전산학과(석사)
1985년~1988년 중앙대학교 대학원 전산학과(박사)
1983년~현재 경상대학교 컴퓨터과학과 교수
관심분야 : 소프트웨어 신뢰성, 시험도구 자동화, 시스템 분석 및 설계, 신경망