20-1-T01

# 힙 메모리 분석 및 검증의 연구 동향

## (A Survey of Heap-Memory Analyses and Verifications)

이 욱 세
한양대학교 컴퓨터공학과
oukseh@hanyang.ac.kr

요 약

힙 메모리에 대한 프로그램 분석은 그 난해성으로 인해 오랫동안 뚜렷한 연구 성과가 없다가 최근 독보적인 연구들로 다시 관심이 집중되기 시작했다. 본 동향 분석은 현존하는 힙 메모리 분석들을 메모리 객체들을 요약하는 방법들로 분류하여 설명한다. 특히, 출생지 기반 분석, 접근경로 기반 분석, 모양분석, 분리논리의 루프불변식 유추 엔진 등을 설명한다. 본 조사를 통하여 연구자들이 새로이 힙 메모리 연구를 시작하는데 도움이 되고자 한다.

## 1. 서 론

힙 메모리(heap memory)에 대한 프로그램 분석은 그 난해성으로 인해 포인터 참조 분석(point-to analysis)[1,16]과 같은 단순한 분석 위주로 연구되어 오다가 최근에 복잡한 프로그램도 검증할 수 있는 기술들이 선보이고 있다. 새로운 장을 열은 연구는 모양분석(shape analysis)[15]과 분리논리(separation logic)[13,14]이다. 두 연구는 프로그램 분석과 프로그램 논리라는 다른 형태로 고안되었으나 모두 복잡한 포인터 프로그램을 자동으로 검증하려는 목적을 가지고 있고, 유사한 수준의 성과를 보이고 있다. 두 연구를 시발점으로 많은 연구들이 뜨겁게 이루어지고 있다.

본 동향 분석은 현존하는 힙 메모리 분석들을 힙 셀(heap cell)들을 요약하는 방법으로 분류하여 설명함으로써 새로이 연구를 시작하는 연구자들이 쉽게 기존 연구를 이해함으로써 새로운 아이디어를 도출하는데 도움이 되고자 한다. 특히, 출생지 기반 분석(allocation-site-based analysis)[2], 접근경로 기반 분석(access-path- based analysis)[7], 모양분석, 분리논리의 루프불변식 유추 엔진 등을 살펴보기로 한다.

## 2. 힙 메모리 분석 및 검증의 난해성

우선 힙 메모리 분석을 자세히 설명하기 전에 힙 메모리 분석의 독특한 난점이 무엇인지 살펴보기로 하자. 그 난해성의 두 축은 (1) 힙 셀을 어떻게 요약하여 식별할 것인가와 (2) 힙 셀의 내용변경이 분석 상에서 정확하게 추적될 수 있는가이다.

## 2.1. 힙 셀의 식별

일반적인 의미구조에서 힙 셀은 주소(address 또는 location)를 통해 실체가 존재하지만 이 주소값은 우리가 힙 메모리를 이해할 때 도움이 되지 않는다. 셀이 생성되면 주소가 할당되고 그 주소로 셀을 접근한다. 문제는 주소값은 이해하고자하는 힙 셀의 속성과는 무관하다는 것이다. 예를 들어 리스트(list)를 생성했을 때 첫 셀의 주소가 105라고 해도, 다음 셀의 주소는 105와는 아무런 관련이 없다. 한 리스트의 원소라는 공통된 속성을 가진 셀들의 주소 사이에는 아무런 연관성이 없는 것이다.

이러한 실상이 없는 주소값 대신에 힙 셀 또는 힙 셀들을 요약해서 부르는 방법이 필요한데, 현존하는 방법을 크게 두 가지 방법으로 분류할 수 있다.

- 고정된 이름을 사용하는 경우: 생성된 장소(allocation site)로 부르거나, 사용자가 미리 지정한 영역(region)으로 힙 셀을 식별하는 방법이다. 고정된 이름을 사용하기 때문에 정적으로 분석하기 쉬운 면이 있으나, 프로그램에서 힙 셀의 관계를 변화시킬 때 그 변화를 추적할 수 없는 단점이 있다. 일반적으로 고정된 이름을 사용하는 경우 저장공간의 주소 또는 이름을 사용하기 때문에 저장기반 의미구조(store-based semantics)를 따르는 분석이라 한다.
- 고정되지 않은 이름을 사용하는 경우: 셀을 가리키고 있는 변수 또는 변수에서부터 그 셀에 도달하는 경로 등과 같은, 프로그램이 실행되면서 변경되는 이름을 사용하여 식별하는 방법이다. 힙 메모리의 변화에 적응할 수는 있으나, 고정되어 있지 않아 분석 중에 다루기가 어려운 편이다. 고정되지 않은 이름을 사용하는 경우, 그 이름은 저장공간과는 아무런 연관이 없고, 단지 다른 저장공간과의 관계에 의해 이름이 구성되기 때문에 저장소없는 의미구조(storeless semantics)를 따르는 분석이라고 부른다.

## 2.2. 완전한 변경 (Strong Update)

힙 메모리 검증에서 정확도를 요하면서도 확보하기 힘든 부분이 힙 셀의 내용변경(memory update)이다. 분석 도중 힙 셀의 내용변경을 정확히 추적하기 어려운 경우를 들면 다음과 같다.

- 포인터의 부정확성: 변경할 셀을 가리키는 변수가 두 개 이상의 셀을 가리킬 가능성이 있을 때, 그 셀들의 내용을 함부로 변경할 수 없다. 왜냐하면, 그 중 어떤 셀은 그 변수가 가리키지 않을 수도 있기 때문이다.
- 힙 셀 요약의 부정확성: 변경할 셀이 두 개 이상의 셀을 요약한 하나의 객체로 요약되었을 때, 그 객체의 내용을 함부로 변경할 수 없다. 이유는 객체가 의미하는 다른 셀의 내용은 변경되지 않기 때문이다.

부정확한 상황에서는 분석의 정확도를 저하시키는 불완전한 변경(weak update)을 할 수밖에 없다. 불완전한 변경이란, 변경될 셀이 다른 셀들과 분별이 안 될 때, 모든 셀의 내용을 변경시키지 못하고, 안전하게 가능성을 추가하는 것이다. 예를 들어, x가 1번 셀 또는 2번 셀을 가리키고 있고, 1번 셀에 3, 2번 셀에 4가 들어 있을 때, “*x=5”하면, 1번 셀에는 3 또는 5, 2번 셀에는 4 또는 5가 들어 있다고 결론짓는 것이다. 변경이 이루어지지 않고, 분석의 안전성을 위해서 가능성을 확장하는 것이다.

완전한 변경(strong update)를 하기 위해서

는 다음과 같은 조건을 만족해야 한다. (1) 변경할 셀을 가리키는 변수가 반드시 그 셀을 요약한 객체만 가리켜야 한다. (2) 변경할 셀을 요약한 객체가 다른 추가의 셀을 포함해서는 안된다. 이 두 가지 조건을 앞으로 소개할 분석 및 검증 방법들이 어떻게 다루는지 주의 깊게 살펴보도록 하자.

## 3. 생성시점 상태를 요약하는 방법론

Deutsch는 주소값이란 힙 셀이 생성되었을 때의 상태라고 단정하고, 그 상태를 요약하여 힙 메모리 분석기를 고안할 수 있다고 하였다 [6]. 이 방법론을 사용한 분석은 출생지 기반 분석(allocation-site-based analysis)[2]과 영역 기반 분석(region-based analysis)[17]이 대표적이다.

### 3.1. 출생지 기반 분석 [2]

힙 셀이 생성되었을 때의 상태를 요약할 때, 생성된 프로그램 지점, 즉, 출생지만으로 요약하는 방법론이 실용적으로 사용되고 있다. 방법론의 핵심은 어떤 프로그램 지점에서 생성된 모든 셀을 그 지점으로 호칭하는 것이다. 비유하자면 한국에서 출생한 사람들을 출생 지방 별로 분류하는 것이다. 그러면 한국 출생 사람이 지방의 개수만큼 유한하게 요약되어 분석이 가능하다.

출생지 별로 메모리 객체를 분류하는 방법의 문제는 셀의 내용을 변경하는 코드에 대해 완전한 변경을 하기 어렵다는 것이다. 예를 들면, 서울 출생의 사람 손에는 모두 500원 동전이 있다고 가정했을 때, 내가 서울 출생의 누구에서 500원 동전을 치우고 100원짜리 동전을 놓은 경우를 생각해 보자. 임의의 서울 출생의 손에 있는 동전은 무엇인가? 안전한 답은 100원 또는 500원이다. 이에 대한 보다 정확한 답을 얻기위한 노력 중 하나가 단일성(uniqueness)을 사용하는 방법이다. 서울 출생의 사람이 단 한 사람일 때는 정확한 답을 낼 수 있다. 이 경우 서울 출생에게 100원을 주면 서울 출생의 모든 사람의 손에는 반드시 100원이 있게 된다. 각 출생지에서 태어난 메모리 객체의 수가 하나 이하인가 하는 정보를 유지함으로써 정확도가 높아질 수 있다. 하나 이하일 때는 완전한 셀 내용 변경이 가능한 것이다. 이러한 방법은 여전히 실용적인 분석기에서 사용되고 있다 [5,10].

그러나 단일성을 사용하더라도 완전한 변경은 좋은 경우에만 가능할 뿐 나쁜 경우에는 불완전한 변경을 통해 분석의 정확도가 저하된다. 특히 재귀적 자료 구조의 경우 완전한 변경이 어려울 확률이 높다. 리스트나 트리와 같은 재귀적 자료구조는 일반적으로 내부의 메모리 객체들이 동일한 출생지에서 생성되게 된다. 그러면 모두 하나로 뭉뚱그려 계산되어 부정확한 정보를 양산하게 된다.

### 3.2. 영역 기반 분석 [17]

영역 기반 분석(region-based analysis)은 출생 시점 대신 출생지에 사용자가 제공한 영역 이름을 사용함으로써 사용자의 의도에 따른 분류를 가능하게 한 방법론이다. 비유해서 말하자면, 청주 출생이든 충주 출생이든 모두 충북 출생이라고 사용자가 프로그램에 첨언함으로써 청주 출생, 충주 출생을 모두 하나의 충북 출생으로 분류해서 분석하는 것이다.

영역 기반 분석이 출생지 기반 분석보다 정확할 수 있는 이유는 영역을 사용한다는 자체보다 영역 다형 함수(region-polymorphic function)를 사용하는데 있다. 어떤 함수 내

부의 특정 코드에서 생성된 힙 셀은 출생지 기반 분석에서는 일반적으로 같은 출생지를 가진다. 영역 다형 함수에서는 함수가 영역 인자를 받을 수 있도록 함으로써, 같은 코드에서 생성된 힙 셀에 다른 영역을 부여할 수 있다. 예를 들어, 함수 f가 영역 인자 ρ를 받아 그 코드에서 생성된 힙 셀의 영역을 ρ라고 한다고 하자. 처음 f를 호출할 때 영역 인자로 r을 넘기면 생성된 셀의 영역은 r이 되고, 다음에 f를 호출할 때 s를 인자로 넘기면 생성된 셀의 영역은 s가 되는 것이다. 즉, 함수 호출 때마다 함수 코드를 다르게 분석하는 문맥 민감 분석(context- sensitive analysis)을 다형 타입 시스템(polymorphic type system)으로 자연스럽게 구현한 것이다.

영역 기반 분석 또한 출생지 기반 분석과 마찬가지로 힙 셀의 출생지에 관련하여 셀을 고정되게 분류하기 때문에 같은 한계성을 가지고 있다. 완전한 변경이 가능한 경우가 적고, 재귀적 자료구조에 적합하지 못하며, 프로그램의 수행 도중 셀을 분류하는 방법이 변경되지 않기 때문에 유연한 힙 메모리 분석은 어렵다.

## 4. 접근 경로 기반 분석

접근경로기반 분석(access-path-based analysis)은 힙 셀의 의미가, 생성되었을 때의 상태에 관계된 것이 아니라 접근 경로에 의해 결정된다는 것에 착안하여 고안된 분석이다. 예를 들면 프로그램에서 리스트는 여러 출생지에서 생성된 셀로 구성될 수 있지만, 그 리스트의 셀들은 특정 변수에서 도달가능하다는 공통점을 가지고 있다. 이런 경우 출생지보다는 그 셀에 도달하는 경로를 가지고 셀의 의미를 이해하는 것이 메모리를 이해하는데 도움이 된다.

### 4.1. Deutsch의 경로기반 분석 [7]

Deutsch가 고안한 경로 기반 분석은 셀을 경로로만 식별하여 분석한다. 예를 들어 어떤 셀이 있는데 그 셀에 3이 저장되어 있으면, 그 셀로 가는 경로 $p$가 3을 갖고 있다고 하는 것이다. 어떤 코드에 의해 그 셀에 도달하는 경로가 변경되었다면, 변경된 경로 $p'$에 3이 있다고 변경한다. 앞서의 출생지 기반 분석과 달리 셀을 식별하는 이름이 $p$에서 $p'$으로 프로그램 지점에 따라 변경되는 것이다. 이러한 경로를 요약하여 최종 분석을 만들 수 있는데, 대표적인 성공 예는 Blanchet의 탈출 분석(escape analysis)[1][3]이다.

경로 기반 분석의 약점은 역시 완전한 내용변경(strong update)이 어렵다는 것이다. 셀의 내용이 변경되면 지역적으로 변경시키는 것으로 끝나는 것이 아니라, 전체적으로 힙 메모리를 요약한 정보의 변경이 이루어 질 수 있다. 예를 들어, 1번 셀을 x가 가리키고, 2번 셀을 y가 가리키는데, 2번 셀의 필드 f가 2번 셀을 가리키고, 필드 g가 1번 셀을 가리킨다면, 경로 x와 경로 y.f.….f.g는 동일 셀을 가리키게 된다. 경로 x의 셀의 내용이 변경되면, 이와 동일 셀을 의미하는 모든 경로에 대해 내용이 변경되어야 하는 것이다. 이런 전체적인 변경은 요약을 통해 분석의 정확도를 저하시킬 소지가 높다. 탈출 분석의 경우 ML에 대해 분석하였고 ML은 셀의 내용변경이 비교적 적기에 성공할 수 있었던 것이다.

1) 프로그램 부분에서 생성된 힙 셀이 그 부분을 벗어나서 사용되는지 감별하는 분석

### 4.2. 포인터 참조 분석 [1,16]

접근경로 기반 분석 중에 가장 널리 쓰이는 분석은 포인터 참조 분석(point-to analysis)이다. 경로 중에 길이가 하나인 것만 정확히 분석하고, 경로가 복잡한 것은 모두 무시하는 분석이다. 경로의 길이가 하나인 것은 변수로만 이루어지기 때문에, 결국 두 변수가 같은 동적 메모리 객체를 가리키고 있는지 알아내는 분석이다. 기법은 두 변수가 같은 객체를 가리킬 수 있는지 모든 변수에 대해서 관계(relation)을 찾아 내는 것이다. 관계를 찾아내는 방법은 “x=y”와 같은 문장이 나오면 x와 y는 동일한 객체를 가리킬 수 있다고 결론짓는 것이다. 이와 같이 단순한 상황에 대해서 분석을 정교하게 하고, 조금 더 복잡한 상황에 대해서는 대충 분석하여 분석의 비용이 비싸지 않도록 고안하였다. 예를 들어 “x=**y” 문장이 나오면 x는 어느 변수하고도 같은 객체를 가리킬 수 있다고 뭉뚱그려 버린다.

포인터 참조 분석은 복잡한 프로그램을 검증할 정도로 정확도는 높지 않지만, 그 비용이 저렴해서 프로그램 최적화 등에 널리 쓰이고 있다. 포인터를 고려하지 않은 다른 프로그램 분석의 입장에서는 포인터 참조 분석으로부터 최소한의 정보를 얻어 분석의 정확도를 높일 수 있기 때문이다. 예를 들어, 상수분석(constant propagation)의 경우, 포인터 변수에 관련된 구문 “*y=1; *z=3; x=*y;” 이 나왔을 때 y, z가 같은 셀을 가리킬 상황을 고려하여 x에 대해 안전한 결론을 내려야 한다. 만약 포인터 참조 분석이 y가 z와 다른 객체를 가리킨다고 정보를 제공하면, x가 1을 가진다고 정확한 결론을 내릴 수 있는 것이다.

## 5. 상황별 분류를 사용한 분석

힙 셀을 식별하는데 있어 접근 경로로 분류하는 것이 항상 이해를 돕는 것은 아니다. 예를 들면, 리스트를 다루는데 있어 일반적으로 리스트의 머리쪽에서 꼬리쪽으로 이동하면서 반복 또는 재귀를 통해 연산을 수행하게 된다. 매 반복마다 리스트의 중간 부분을 지역적으로 보면서 연산을 수행하는데, 이 때 현재 보고 있는 셀, 이미 지나간 셀들, 앞으로 지나갈 셀들 간에 분류가 필요하다. 이러한 분류를 경로로는 자연스럽게 해결되지 못한다.

최근의 동향은 힙 셀들의 분류를 상황에 따라 달리하는 방법론들이 연구되고 있다. 예를 들면, 그래프 노드, 가상 주소(symbolic location), 임시 변수(auxiliary variable) 등을 사용하여 힙 셀들을 분류하는 것이다. 이러한 분류 기준은 고정된 것이 아니라 분석 도중에 분류 기준이 변경되는 유연한 것이다.

### 5.1. 모양분석 [15]

모양분석(shape analysis)은 메모리를 그래프로 모델링하여 셀의 내용변경이 완전하게 이루어지는데 초점을 맞춘 분석이다. 셀을 식별하는 것은 출생지도, 경로도 아닌, 그래프의 노드(node)이다. 노드는 여러 셀을 요약한 객체이며, 셀이 생성되면서 새로 만들어 지기도 하고, 다른 노드들과 합쳐지기도 하고, 셀을 사용하면서 다시 생성되기도 한다. 노드의 이름은 아무런 의미도 없고 노드의 속성에 따라 의미가 부여된다.

완전한 내용변경이 가능한 핵심 이유는 구체노드(concrete node), 즉, 셀 하나를 의미하는 노드를 사용하기 때문이다. 앞서 기술했듯이 완전변경의 조건 중 하나는 변경

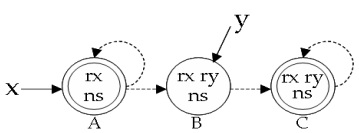
을 요하는 요약된 객체가 의미하는 셀이 하나이하라는 것이다. 모양 분석에서 셀은 두 분류로 나누어 요약된다. 구체노드와 셀 여러 개를 의미하는 정리노드(summary node)이다. 셀이 생성되면 구체노드로 표현되지만 다른 정리노드와 속성이 같아지면 그 정리노드로 편입된다. 그러다가 다시 그 셀의 내용을 변경하려고 할 때는, 정리노드에서 그 셀을 꺼내 구체노드로 만든 후 완전하게 그 내용을 변경시킨다. 이렇게 정리노드에서 구체노드를 꺼내오는 작업을 초점맞추기(focus)라고 부른다.

완전한 내용변경이 가능한 또 다른 핵심 이유는 가능한 경우를 하나로 요약하지 않고 집합으로 다룬다는 것이다. 변수 두 개가 동일 셀을 가리킬 수도 있고 아닐 수도 있는 경우, 두 경우를 나누어 분석함으로써, 각각의 경우를 정밀하게 분석할 수 있는 것이다. 집합으로 다룸으로써 분석 비용은 증가하지만 메모리를 정리하여 경우의 수를 줄여 무한히 경우가 늘어나지는 않게 고안하였다.

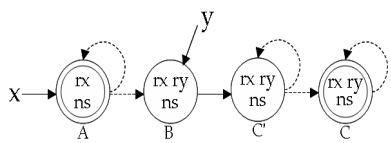
왜 정교한 분석이 가능한지 예를 통해 보도록 하겠다. 다음은 x가 리스트를 가리키고 있었다면 y가 리스트를 따라 끝까지 움직이는 코드이다.

```
y=x; while(y!=nil) y=*y;
```

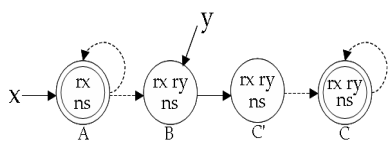
y가 리스트의 중간을 가리키는 경우에서부터 설명을 하도록 하겠다. 그 경우 다음과 같은 그래프로 힙 메모리를 모델링한다.



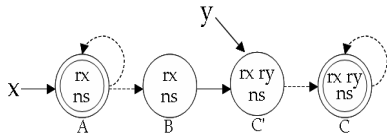

그림에서 겹원은 정리노드, 원은 구체노드를 의미하고, 점선은 포인터가 있을지 없을지 모른다는 뜻이다. 읽어보면, x는 A내의 셀을 가리키고, A내의 셀 다음(next)은 A내의 셀이거나, B셀이거나, nil이다. y는 B셀을 가리키고, B의 다음은 C내의 셀이거나 nil이다. C내의 셀의 다음은 C내의 셀이거나 nil이다. 원안에 적힌 rx는 x로부터 도달가능, ry는 y로부터 도달가능, ns는 셀을 가리키는 포인터가 한 개 이하를 의미한다. “y=*y”를 실행할 때 y의 다음 셀에 초점을 두어야 한다. 가능성은 *y가 nil일 때와 아닌 경우이다. *y가 nil이 아닌 경우만 살펴보자. 아닌 경우 정리노드 C에서 구체노드 C′을 꺼낸다. 구체노드 C′은 C의 속성을 모두 만족한다.




초점맞추기를 할 때 가지치기 작업이 필요한데, 이는 각 노드의 속성에 따라 불필요한 불확실성을 제거하는 것이다. 예를 들어 C′에서 C′으로 가는 포인터는 불가능하다. 왜냐하면 그 포인터가 있으면 C′이 ns 속성을 만족한다는 것에 위배되기 때문이다. 정리하면 다음과 같은 그래프가 된다.




여기서 “y=*y”를 수행하면 다음과 같이 변경된다.




포인터가 변경될 때 B 노드처럼 속성이 변경될 수 있다. 최종적으로 A노드의 속성과 B노드의 속성이 같으므로 합쳐줄 수 있다. 합치고 나면 시작 때와 같은 그래프가 된다.

예제 과정을 통해 주목할 부분은 값을 변경할 셀은 항상 초점맞추기를 통해 구체노드로 만든다는 것이다. 구체노드이기 때문에 완전한 변경이 가능하다.

모양분석은 매우 정밀한 분석이긴 하지만 비용이 비싸고 사용자가 제공할 것이 많은 약점이 있다. 최근에 모양분석은 증명하기 까다로운 프로그램인 DSW 알고리즘의 옳음을 자동으로 증명해 낼만큼 그 정밀도가 높음을 보여 주었다 [12]. 그러나 그 모든 것이 완전 자동이라고 하기에는 사용자가 제공해야 할 정보가 많다. 관심있는 속성과, 속성에 따라 어떻게 가지치기가 되는지, 속성은 어떻게 변경되는지 수동으로 기술해 주어야 한다. 게다가 커다란 프로그램에 적용하기에는 이론적으로나 실험적으로 복잡도가 높아 해결해야 할 과제가 많이 남아있다.

## 5.2. 분리 논리 [13,14]

분리 논리(separation logic)는 힙 메모리를 다루는 프로그램을 검증하기 위한 프로그램 논리 시스템이다. 기존의 논리식에 힙 메모리를 다루기 좋은 스타 연산자 *를 추가하였다. 논리식 “P*Q”의 의미는 힙 메모리의 한 부분은 P를 만족하고, 한 부분은 Q를 만족하는데, 그 두 부분은 겹침이 없고, 그 두 부분외에 추가로 사용되고 있는 힙 메모리는 없다라는 뜻이다. 이런 분리 논리식에관해 Hoare 트리플(triple)을 유추하기 위한 논리 시스템이 분리논리이다.

분리 논리가 각광받고 있는 이유는 힙 셀을 검증하는데 있어 매우 중요한 두 속성을 잘 표현하고 있기 때문이다.

- 겹침이 없음 (separation)
- 지역적인 힙 메모리의 변경이 다른 힙 메모리에 영향을 주지 않음 (frame rule)

이러한 속성을 쉽게 표현할 수 있어서 분리논리를 사용해 DSW 알고리즘의 옳음을 적은 노력을 들여 수동으로 증명할 수 있었다.

분리 논리 자체는 자동 메모리 검증 및 분석기가 아니지만, 분리 논리 규칙에 맞추어 프로그램 부분에서 만족하는 논리식을 찾아내는 논리식 유추기가 개발, 연구되고 있다. 이런 유추기의 핵심은 루프(loop)에서 루프불변식을 얼마나 정확히 찾느냐에 달려있다.

유추 엔진의 첫 번째 사례는 문법기반 모양분석[11]이다. 문법기반 모양분석은 모양분석과 유사한 엔진위에 정리노드를 문법형태를 사용한 방법론으로 이항힙구조(binomial heap) 정도의 복잡한 자료구조를 생성하는 코드에 대해 완전자동으로 루프불변식을 유추해 낸다. 자세한 사항은 다음 절에서 설명하도록 하겠다. 문법기반 모양분석 외에 분리논리 불변식 유추 연구는 단순한 리스트와 같은 자료구조에 대해, 보다 복잡한 속성을 유추하거나[8] 포인터 연산도 다룰 수 있도록[4], 정리증명기(theorem prover)를 동원하여 유추하는 방법들이 연구되고 있다.

## 5.3. 문법기반 모양분석 [11]

문법기반 모양분석은 모양분석에서 정리노드의 속성을 문법으로 기술하는 방법론이다. 문법으로 기술하게 된 동기는 다음과

같다. (1) 사용자가 증명하고자 하는 자료구조에 따라 적절한 속성을 사용자가 제공해야 하는데 이를 문법으로 대체하여 자동화하자는 것이다. (2) 문법을 사용하면 초점맞추기가 상대적으로 단순하여 분석 비용의 절감과 분석 정확도 향상에 도움이 될 것이라 판단되었다.

문법은 일반적으로 사용하는 정규적인 자료구조를 표현할 수 있다 [9]. 예를 들어 리스트는 "List ::= nil | ○→List"와 같이 표현할 수 있다. 여기서 ○는 한 칸짜리 셀을 의미하고 그 내용은 포인터(→)인데 List가 의미하는 자료구조를 가리키고 있다는 뜻이다. 비단말기호(non-terminal)에 인자를 사용하여 보다 복잡한 자료구조까지 표현할 수 있다. 예를 들어 "Lseg(a) ::= nil | ○→Lseg(a) | ○→a"는 리스트의 조각으로 그 마지막 셀이 인자 a를 가리키고 있다는 뜻이다.

앞서의 예제를 문법기반 모양 분석을 통해 다시 분석해 보자. 동일한 상태에서 시작하도록 하겠다.

x→Lseg(n), y→n→List

여기서 Lseg, List 등은 앞 문단에서 설명한 것과 같은 의미로, 사용자가 미리 제공한 것이 아니라 자동으로 유추된 것이다. x는 끝이 n인 리스트 조각을 가리키고, y는 n 셀을 가리키고, n 셀의 다음은 리스트이다. "y=*y"를 분석하기 위해 y의 다음 셀에 초점을 두면 List에서 셀 하나를 꺼내 와야 한다. List의 정의에 따르면 List는 nil이거나 ○→List 이기 때문에 다음과 두 경우가 생긴다.

x→Lseg(n), y→n→nil

x→Lseg(n), y→n→m→List

두 번째의 경우 꺼낸 셀(○)에 새로운 이름 m을 주었다. 두 번째 경우만 고려해서 "y=*y"를 수행하면 다음과 같이 변경된다.

x→Lseg(n), n→m, y→m→List

루프의 마지막에 펼쳐진 셀들을 정리를 하는데 고안된 휴리스틱을 적용하면 "Lseg(n), n→m"은 Lseg(m)으로 합쳐질 수 있다. 결국 최종 결론은 다음과 같이 처음상태와 같아진다.

x→Lseg(n), y→n→List

예제에서 살펴본 것과 같이 문법기반 모양분석은 원래의 모양분석에 비해 상대적으로 초점맞추기가 단순하다. 초점맞추기가 단순히 문법 정의를 펼치는 것으로 완료되기 때문에 불확실성 때문에 발생하는 가지치기 비용과 분석 정확도 손실을 염려할 필요가 없는 것이다.

문법기반 모양분석의 약점은 문법으로 표현되지 않는 정규적이지 못한 구조는 다룰 수가 없다는 것이다. 모양 분석은 복잡한 구조일 경우에도 각 셀들의 속성으로 부정확하지만 요약이 가능한데, 문법기반 모양분석은 셀들이 문법으로 정리가 되지 않으면 계속 남아 분석을 불가능하게 만들 수 있다. 예를 들어, 임의의 그래프는 현재 문법으로 표현할 수가 없기 때문에 임의의 그래프를 생성하는 코드를 분석하면 아무런 결론도 내리지 못한다.

## 6. 결론

힙 메모리 분석 및 검증에 대한 연구는 많은 진전을 이루었지만 아직도 가야할 길

은 멀다고 판단된다. 앞으로의 연구 방향을 크게 두 가지로 나누어 전망할 수 있는데, 하나는 저렴하면서도 정확한 힙 메모리 분석이고, 다른 하나는 현재의 기술보다도 정밀한 힙 메모리 검증이다.

현재의 힙 메모리를 고려하지 않는 프로그램 분석들은 포인터 참조 분석과 같이 매우 저렴하면서 부정확한 분석을 사용하고 있어 분석의 결과에 악영향을 끼치고 있다. 현재까지 힙 메모리 분석의 연구결과를 바탕으로 포인터 참조 분석을 대체할 수 있는 저렴하면서도 좀 더 정확한 메모리 분석이 고안될 수 있을 것으로 보인다. 이는 힙 메모리 분석의 실용화를 위해 매우 중요한 연구 방향으로 판단된다.

또 다른 방향은 지금보다도 더 정밀한 힙 메모리 분석을 개발하는 것이다. 안전중심 프로그램(safty-critical program)의 경우 힙 메모리 오류로 인해 그 안전성을 해칠 위험에 노출되어 있다. 지금보다 더 정밀한 힙 메모리 분석 또는 검증기를 개발하여 비싸더라도 그 안전성을 보장받을 수 있다면 소프트웨어로 인해 발생하는 여러 위험요소들을 제거하는데 도움을 줄 것이라고 판단된다.

## 참 고 문 헌


[1] L. O. Andersen. *Program Analysis and Specialization of the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[2] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 296-310, 1990.

[3] B. Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 25-37, 1998.

[4] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic. In *Proceedings of the International Symposium on Static Analysis*, pages 182-203, 2006.

[5] T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh. A Practical String Analyzer by the Widening Approach. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, November 2006.

[6] A. Deutsch. On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher-Order Functional Specifications. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 157-168, 1990.

[7] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 230-241, 1994.

[8] D. Distefano, P. O'Hearn, and H. Yang, A local shape analysis based on separation logic. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems*, pages 287-302, 2006.

[9] P. Fradet and D. Le Métayer. Shape types. In *Proceedings of the ACM Symposium on Principles of Programming Languages,* pages 27-39, January 1997.

[10] Y. Jung, J. Kim, J. Shin, and K. Yi. Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis. In *Proceedings of International Symposium on Static Analysis*, pages 203-217, September 2005.

[11] O. Lee, H. Yang, and K. Yi. Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis. In *Proceedings of the European Symposium on Programming*, pages 124-140, April 2005.

[12] A. Loginov, T. Reps, and M. Sagiv. Automated Verification of the Deutsch-Schorr-Waite Tree-Traversal Algorithm. In *Proceedings of International Symposium on Static Analysis*, pages 261-279, 2006.

[13] P. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 268-280, January 2004.

[14] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 55-74, July 2002.

[15] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217-298, 2002.

[16] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 32-41, 1996.

[17] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109-176, 1997.

이 욱 세

1991년~1995년 한국과학기술원 전산학과(학사)

1995년~1997년 한국과학기술원 전산학과(석사)

1997년~2003년 한국과학기술원 전산학과(박사)

2004년~현재 한양대학교 컴퓨터공학과 교수

관심분야 : 프로그램 분석 및 검증, 메모리 분석, 타입 시스템, 함수형 언어, 컴파일러