자바에서의 Null을 할당할 수 있는 필드를 찾기 위한 데이터 흐름 분석

김민균° 이숙희 권용래
한국과학기술원 전자전산학과
{mkkim°, shlee, kwon}@salmosa.kaist.ac.kr

Data-flow Analysis for Finding Null-assignable Fields in Java

Minkyoon Kim°, Sukhee Lee, Yong-Rae Kwon
Dept. of Electronic Engineering and Computer Science, KAIST

요 약

이 논문에서는 자바 프로그램이 실행 중에 사용하는 힙(heap)의 크기를 줄일 수 있는 데이터 흐름 분석 기법을 제안한다. 이 알고리즘은 클래스를 분석하여, 사용될 때마다 새로 정의되는 필드(field)들을 찾는다. 이 필드의 마지막 사용 후에 null 값을 필드에 할당하면 필드가 가리키고 있던 객체를 더욱 빨리 회수할 수 있게 되고, 이로 인해 객체들이 차지하는 힙 공간을 줄일 수 있다. 이 알고리즘은 private 필드만을 대상으로 분석을 수행한다. 우리의 궁극적인 목표는 이 알고리즘을 확장하여 모든 필드들을 분석하고, 힙 사용량을 줄이기 위해 null을 할당하도록 바이트 코드를 자동으로 수정하는 기법을 개발하는 것이다.

1. 서론

그림 1은 객체의 생성부터 객체가 Garbage Collector(이하 GC)에 의해 회수될 때까지의 단계를 보여 준다. 객체가 생성된 후 처음으로 사용될 때까지의 시간을 래그 타임(lag time)이라 하고, 객체의 마지막 사용 후 GC에 의해 회수될 때까지의 시간을 드래그 타임(drag time)이라고 한다[5]. 객체가 드래그 상태에 있으면 그 객체는 불필요하게 힙 영역을 차지하기 때문에 결국 프로그램의 메모리 사용량이 늘어나게 된다. [2]에 따르면 SPECjvm98 벤치 마크 프로그램들에 대해, 객체들을 마지막 사용 직후 회수할 수 있다면 23%에서부터 74%까지의 메모리 사용량을 절약할 수 있다고 한다. 특히 메모리가 제한된 시스템에서 동작하는 자바 가상 머신(JVM)의 경우 드래그 타임으로 야기되는 메모리 사용량의 증가는 심각한 문제가 될 수 있다.

드래그 상태에 있는 객체들을 빨리 회수하기 위한 방안으로 다음과 같은 방법들이 제안되었다. [1]에서는 힙 프로파일링(heap-profiling) 도구를 이용하여 드래그에 대한 정보를 수집한 후, 프로그래머가 그 정보를 바탕으로 코드를 재작성하는 방법을 기술하였다. 하지만 이 방법은 힙 프로파일링을 해야 하고, 프로그래머가 직접 코드를 재작성해야 하므로 많은 시간과 노력이 요구된다. 또 프로그래머의 코드 수정 중 또 다른 버그가 생길 가능성도 있다.

다른 방법으로는 참조 변수들(reference variables)의 liveness 정보를 이용하는 것이다. liveness 정보를 이용하여 객체들을 더 빨리 회수할 수 있다는 것은 잘 알려 진 사실이다. [4]에서는 GC가 지역 참조 변수들의 프로시저-내(intraprocedural) liveness 분석을 이용하는 것에 대하여 실험을 행하였다. 실험 결과 대부분의 프로그램의 경우 GC가 지역 변수들의 liveness 분석 결과를 이용하여도 메모리 사용량을 거의 줄일 수 없었다. [3]에서는 지역 참조 변수와 정적 참조 필드, 객체 참조 필드의 liveness 분석을 통해 얻을 수 있는 메모리 절감 효과를 예측하는 실험을 수행하였다. 실험 결과, 큰 메모리 절감 효과를 얻기 위해서는 정적 참조 필드들과 객체 참조 필드들의 프로시저-간(interprocedural) liveness 분석이 필요하다. 하지만 우리는 정적 참조 필드와 힙 참조 필드들의 liveness 정보를 구하는 정적 분석(static analysis) 알고리즘에 대한 연구 결과는 찾을 수 없었다.

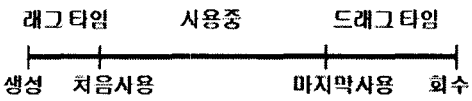

그림 1. 객체의 일생 ([4]에서 발췌)

본 논문에서는 private 참조 필드들 중 참조 값을 필요 없이 유지하는 필드들을 찾는 데이터 흐름 분석 알고리즘을 제안한다. 알고리즘을 통해 찾은 필드들에 null 값을 할당하면 프로그램이 수행 중에 사용하는 메모리를 절약할 수 있다. 필드의 마지막 사용 정보를 가지고 있도록 알고리즘을 확장하면, 필드에 null을 할당하도록 바이트 코드를 자동으로 수정할 수 있다.

2. 알고리즘

본 장에서 기술하는 순방향 데이터 흐름 분석을 통하여 우리는 null을 할당하여 메모리를 절약할 수 있는 참조 타입의 private 필드들을 알아낸다. 하나의 클래스를 입력으로 받는 이 알고리즘을 수행하면 사용될 때마다 새로운 값으로 재정의되는 필드들을 결과값으로 준다. 알고리즘 기술에 필요한 용어들은 그림 2와 같이 정의한다.

def x: 필드 x에 값을 할당 바이트 코드
use x: 필드 x의 값을 사용하는 바이트 코드
call C: 메소드 C를 호출하는 바이트 코드
M(A): 클래스 A에 정의된 모든 메소드들의 집합
NPM(A): 클래스 A에 정의된 non-private 메소드들의 집합
allpath(m): 메소드 m의 모든 실행 경로들의 집합

그림 2. 알고리즘 기술에 필요한 용어 정의

조건 2.1
∀ 메소드 m∈NPM(A), ∀p∈allpath(m), 실행 경로 p에 존재하는 모든 use x를 dominate하는 def x가 존재한다.

클래스의 private 필드에 접근하려면 해당 클래스가 제공하는 non-private 메소드들을 이용해야 한다. 따라서 임의의 클래스 A의 private 필드 x가 조건 2.1을 만족한다는 것은 클래스 A의 non-private 메소드들이 x를 전혀 사용하지 않거나, x를 사용할 때 마다 새로운 값으로 정의하여 사용한다는 것을 의미한다. 그러므로 마지막 use x 후에는 x의 값을 유지할 필요가 없다. 마지막 use x후에 x를 null로 정의하여 x가 가리키던 객체로의 포인터를 끊으면, GC는 x가 가리키던 객체를 회수할 가능성을 갖게 된다. 그림 3은 조건 2.1을 만족하는 클래스의 예를 나타내고 있다.

2.1절에서는 한 메소드 내에서 분석하는 방법을 구체적으로 살펴보고 2.2절에서는 메소드간의 호출을 고려하여 프로시저-간(interprocedural) 분석 방법을 기술한다.

2.1. 프로시저-내(Intraprocedural) 분석

본 분석에 사용한 데이터 흐름 정보(data flow information: 이하 DFI) R은 다음과 같이 정의된다.

$$R \subseteq \mathcal{P}(F \times FS)$$

F는 클래스 상에 정의된 reference 타입의 private 필드들의 집합이다. FS(field status)는 각 필드들이 정의(define)되고 사용(use)되는 방식을 나타내고 FS={ddu, udd, none}로 정의된다. FS의 원소들은 ddu π none π udd 의 순서(ordering)를 가지고 각 원소들이 갖는 의미는 아래와 같다.

메소드 내의 임의의 포인트 pt에서의 DFI를 R_{pt}라고 하자.

- (x, ddu)∈R_{pt}이면 메소드의 시작부터 pt까지의 모든 경로가 다음의 조건 중의 하나를 만족한다.
  - i) use x없이 def x만 존재한다.
  - ii) 첫 번째 use x전에 def x가 존재한다. (def dominates uses)
- (x, udd)∈R_{pt}이면 메소드의 시작부터 pt까지의 경로 중에서 아래의 조건을 모두 만족하는 path가 존재한다.
  - i) use x가 존재한다.
  - ii) 첫 번째 use x전에 def x가 존재하지 않는다. (use dominates defs)
- (x, ddu)∉R_{pt}이고, (x, udd)∉R_{pt}이면 (x, none)∈R_{pt}이다.

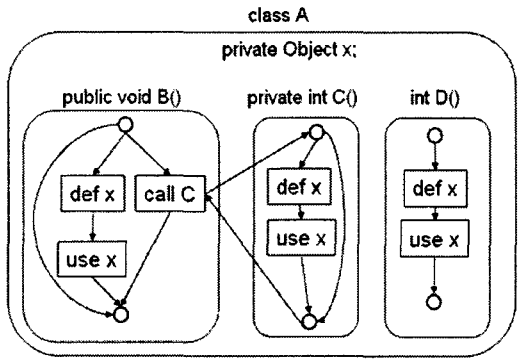

그림 3. 조건2.1을 만족하는 예제. 클래스 A내에 정의된 메소드들의 제어 흐름 그래프. 각 메소드의 마지막 use x 후에 x=null을 추가할 수 있다.

이 문제를 위한 데이터 흐름 식(data flow equation)은 아래와 같이 정의된다.

$$IN(s) = \bigsqcup_{p \in pred(s)} OUT(p)$$

$$OUT(s) = (IN(s) - KILL(s, IN(s))) \sqcup GEN(s, IN(s))$$

함수 IN과 OUT은 프로그램의 각 바이트 코드를 DFI R로 매핑시킨다. S는 바이트 코드들의 집합이다.

$$IN, OUT : S \rightarrow R$$

함수 IN은 아래와 같이 초기화 시킨다.
∀s∈S, IN(s) = {(f, none) | f∈F}
GEN 함수와 KILL 함수는 다음과 같이 정의된다.

$$GEN, KILL : S \times R \rightarrow R$$

이 분석에 영향을 끼치는 자바 instruction들은 필드를 접근하는 instruction인 putfield, putstatic, getfield, getstatic이다. 이 instruction들에 대해 다음과 같이 GEN 함수와 KILL 함수를 정의한다.

s∈S 에서,
i) s가 putfield x 또는 putstatic x 일 때:
  (x, none)∈R 일 경우,
    GEN(s, R) = {(x, ddu)}
    KILL(s, R) = {(x, none)}
  (x, none)∉R 일 경우,
    GEN(s, R) = KILL(s, R) = ϕ
ii) s가 getfield x 또는 getstatic x 일 때:
  (x, none)∈R 일 경우,
    GEN(s, R) = {(x, udd)}
    KILL(s, R) = {(x, none)}
  (x, none)∉R 일 경우,
    GEN(s, R) = KILL(s, R) = ϕ
iii) 그 밖의 경우:
  GEN(s, R) = KILL(s, R) = ϕ

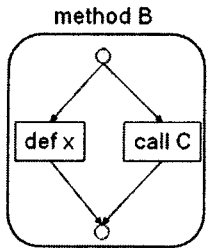

그림 4. 프로시저-간 분석의 설명을 위한 예. 메소드 C의 분석 결과에 따라 메소드 B의 분석 결과가 달라진다.

DFI R_1과 R_2가 다음과 같이 정의되었다고 하자.
$R_1 = \{(f_1, fs_{11}), (f_2, fs_{12}), \cdots, (f_n, fs_{1n})\}$,
$R_2 = \{(f_1, fs_{21}), (f_2, fs_{22}), \cdots, (f_n, fs_{2n})\}$

$R_1 \sqcup R_2$ 는 다음과 같이 정의된다.
$R_1 \sqcup R_2 = \{(f_1, fs_{11}), (f_2, fs_{12}), \cdots, (f_n, fs_{1n})\}$
$\sqcup \{(f_1, fs_{21}), (f_2, fs_{22}), \cdots, (f_n, fs_{2n})\}$
$= \{(f_1, fs_{11} \sqcup fs_{21}), (f_2, fs_{12} \sqcup fs_{22}), \cdots, (f_n, fs_{1n} \sqcup fs_{2n})\}$
$fs_1, fs_2 \in FS$ 에 대해 $fs_1 \sqcup fs_2$는 위에서 명시한 FS의 원소들 간의 순서에 의해 다음과 같다.
$\forall fs \in FS$, $fs \sqcup duu = fs$, $none \sqcup none = none$, $fs \sqcup udd = udd$

2.2. 프로시저-간(Interprocedural) 분석

정의 2.2 임의의 메소드 m의 제어 흐름 그래프를 CFG(m)라고 하자. result(m)은 OUT(CFG(m)의 exit 노드)로 정의한다.

그림 4에서 보면 result(C)의 값에 따라 result(B)의 값이 결정된다. 즉, (x, udd)∈result(C) 이면 (x, udd)∈result(B)이고, (x, duu)∈result(C) 이면 (x, duu)∈result(B)이다. 또 (x, none)∈result(C) 이면 (x, none)∈result(B)이다. 이와 같이 임의의 메소드 m의 result(m) 값을 구하기 위해서는 메소드 m에서 호출하는 다른 메소드들에 대한 분석 정보를 필요로 한다. 이 경우 result(m)을 구하는 방법으로는 다음 두 가지가 있다.

- 메소드 m을 분석하기 전에 메소드 m에서 호출하는 다른 메소드들을 미리 분석하여, 메소드 m을 분석할 때 모든 정보가 미리 준비되어 있도록 한다.
- 메소드 m을 분석할 때 m에서 호출하는 다른 메소드들에 대한 분석 결과가 없을 경우, 해당 메소드 호출을 무시하고 분석을 계속 진행한다. 메소드 m에서 호출하는 메소드에 대한 분석 정보가 후에 계산되었을 때 메소드 m을 다시 분석한다.

첫 번째 방법은 호출 그래프(call graph)에 사이클(cycle)이 존재하는 경우를 처리할 수 없다. 그러므로 본 논문에서는 두 번째 방법을 선택하였다.

분석 결과에 영향을 끼치는 자바 instruction들에 필드를 접근하는 instruction이외에도 메소드 호출 instruction이 포함되어야 한다. s가 메소드 n를 호출하는 statement일 때 GEN 함수와 KILL 함수는 다음과 같이 정의된다.

```
worklist ← the set of all nodes in the call graph
while worklist is not empty
        remove a node n from worklist
        compute result(n)
        if result(n) changed then
                worklist ← worklist U successor(n)
```

그림5. 클래스 내의 메소드들을 분석하는 worklist iterative 알고리즘

i) result(n)이 존재한다면:
GEN(s, R) = {(x, fs) | (x,fs)∈result(n) ∧ (x, none)∈R}
KILL(s, R) = {(x, none)| (x,fs)∈result(n) ∧ (x, none)∈R}
ii) result(n)이 존재하지 않는다면:
GEN(s, R) = KILL(s, R) = ϕ

메소드들을 임의의 순서대로 분석할 수 있다. 어떤 메소드의 분석 결과가 바뀌었다면 이 메소드를 호출하는 다른 메소드들도 다시 분석하도록 하는 worklist iterative 알고리즘[6]을 사용한다.

모든 메소드들에 대한 분석이 끝났다면, 마지막 use 후에 null을 할당할 수 있는 필드들의 집합 FinalResult는 다음과 같다.
FinalResult= {x | ∀m∈NPM(A), (x, null)∈result(m) ∨
(x, ddu)∈result(m) }

3. 결론 및 향후 과제

본 논문에서는 드래그 상태에 있는 객체들을 빠르게 회수할 수 있도록 null을 할당할 수 있는 필드들을 알아내는 데이터 흐름 분석 기법을 제안하였다. 본 논문에서 제안한 알고리즘은 private 필드만을 분석 대상으로 하고 있지만, non-private 필드도 분석 대상에 포함시키기 위한 연구가 현재 진행 중이다. 또 null을 할당하는 코드를 자동으로 바이트 코드에 삽입해 주는 컴파일 시간 최적화 기법도 연구 중이다.

4. 참고 문헌

[1] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap Profiling for Space-Efficient Java. In *Proc. of Prog. Lang. Design and Impl*, pages 104-113, June 2001.
[2] R. Shaham, E. K. Kolodner, and M. Sagiv. On the Effectiveness of GC in Java. In *Proc. of Intl. Symp. on Memory Management*,, pages 12-17, October 2000.
[3] R. Shaham, E. K. Kolodner, and M. Sagiv. Estimating the Impact of Heap Liveness Information on Space Consumption in Java. In *Proc. of Intl. Symp. on Memory Management*, pages 64-75, June 2002.
[4] O. Agesen, D. Detletfs. and E. Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In *Proc. of Prog. Lang. Design and Impl.*, pages 269-279, June 1998.
[5] N. Rojemo and C. Runciman. Lag, drag, void and use - heap profiling and space-efficient compilation revisited. In *Proc. of Intl Conf. on Functional Programming*, pages 34-41, 1996.
[6] G. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194-206, October 1973.