UGRC

Possible ways an object can escape scope :

- Assigning to global/class variable (object aliveness is equivalent to global/class variable) - No Algo to determine for stack allocation
    - Class - UserDatabase
    - Global -
- Storing in Data Structure (List, Map, Set, etc)
- Return Calls (moving to another method scope)
- Argument to another method (similar to return)
- Passing to external objects (object aliveness is equivalent to external object aliveness)

Inferences for last usage and stack allocation for objects - Changed Perspective

- For objects which get referenced by global/class variables, we cannot determine the last usage of the object wrt method.
- This can be in regard to Storing in Data Structure, Another object referring to it.
- When an objects lifetime is matched to a global variable, we can't do the necessary algo.

29 Aug 2023

- Searched for papers on object liveness, liveness analysis with garbage collection and related topics, most relevant papers :.

- Went over a few papers from IITK and IITB by Prof. Amey Karkare pdf

    - Most of the papers involved complete flow-sensitive analysis of the program.
    - But heavy trade off in time for memory management.

- Important paper with similar points-to and them liveness analysis and good results - pdf

    - Region based analysis with heap objects.
    - Deallocation of all objects in a region at once.
    - Given an input Java program, our system produces an output program augmented with region annotations.
    - Annotations include region creation, region deallocation of all objects, and region passing.
    - Flow-insensitive, context-sensitive pointer analysis to partition memory into regions and compute points-to information.
    - Flow-sensitive inter-procedural analysis to determine the region of each object at each program point.
    - Can correlate the region to stack allocation for some instances of region creation.

- Other papers which I will go over for understanding other implementations of flow analysis and garbage collection, not entirely sure if necessary.

    - Extensible intraprocedural flow analysis at the abstract syntax tree level

    - Garbage Collection and Local Variable Type-Precision and Liveness in JavaTM Virtual Machines

- Installed and working with soot but with a few issues, will resolve them or try Sootup(Soot successor).

5 Sep 2023

- Most papers have improved on Escape Analysis, Stack Allocation, and Garbage Collection, but not on Liveness Analysis.
- Went over multiple possble papers on the current issue, most relevant below.
- Heap Profiling for Space-Efficient Java
    - Analysis of drag time and possible solutions.
    - Drag time = product of size of object and time the object is reachable and not alive.
    - 2 part analysis of the program :
        - Analysis of objects and stored into a file
        - Analyse the file producng list of allocatio sites sorted wit potential of drag reduction
    - 3 Optimizations
        - Assigning objects to null
        - Dead Code removal
        - Lazy Allocation
    - Near equal running times but large space optimisation.
    - Main problem, not algorithm to decide null points but is a prior run to decide the null points.
- Estimating the impact of heap liveness information
    - Improvement of the above tool.
    - Requires 1 run of the program to get the liveness information.
    - 1 run marks all points of importance(using stack, static, or heap reference) and are categorized.
    - Then Analysis of liveness takes all reference points as possible null assignment points.
    - At any point the object is used, all prev points are unmarked as possible null assignment points.
    - Last remaining points are the null assignment point.
- Other papers read :
    - Garbage Collection and Local Variable Type-Precision and Liveness in $Java^{TM}$ Virtual Machines
    - Lag, drag, void
    - Liveness-Garbage Collector, Func Lang

Next Work

- One possible paper describing clear analysis method, to be checked for validity: IITK Thesis

- Other possible alternatives (both by Sigmund Cherem, Cornell) :
    - Region Analysis and Transformation for Java Programs
    - Compile-time deallocation of individual objects
        - Just a further implementaion of the above
- Soot Compiler

19 Sep 2023

- Most work primarily surrounding Region based analysis by Tofte, Birkedal, Talpin
- Built on Functional Language ML
- Well typed expression transformed to region based
    - TE < e -> e', (T, p), phi; phi - supersets of all regions required for eval of e
- Functions modified to take run time parameters containing information based on regions.
- Problems
    - Polymorphic Recursion
    - Poor Results for nested and recursive
    - Storage mode allocation
- Bounding regions by analysis
    - Upper bound of # of values in a given region
    - Finite -> Activation Record
    - Infinite -> Linked List of fixed size region pages
- Stats
    - 10x to 0.25x in speed
    - 0.08x to 3000x in storage
    - If modified to be region friendly, much better performance
- Introduced Region Profiling for data leaks and modules for large pieces of code
- Garbage collection
    - Customized Cheney's Algo
    - Applied to each region page which are connected to each other
    - All pages connected in region manner will get copied to to-space when 2/3 of the space of the free list is completed.
    - GC + region interface is worse than just RI
    - But it is better than GC individually(without RI).

26th Sep 2023

- Went over the Cornell paperlink

- The summary of the paper:
    - Region based analysis based on points-to analysis.
    - Stack allocation on non-escaping objects.
- Stats
    - The static analysis takes up 16% of total compilation time(including soot and i/o of files).
    - Wrt memory management, the analysis is slightly worse than a dynamic GC on average.
    - For applications which use significantly less memory, RA is significantly better than GC as GC is set to be called at higher memory intervals.
    - For short lived memory regions RA > GC
    - For long lived memory regions GC > RA
    - For most programs RA < GC by small factor.
- The analysis is divided into 3 parts: Points-to Analysis, Region Liveness Analysis and Region Translation.
- Points-to Analysis modified in format of Region based analysis creates segments the program into regions using flow-insensitve, context-sensitive analysis.
- Region Liveness Analysis is a flow-sensitive, inter-procedural analysis to determine the region of each object at each program point.
- Region Translation is a translation of the program to include region annotations for the final piece of code
- Stack allocation is also included similar to previous work on escape analysis, but with a few changes.
- There is a system of bounded allocations where between the creation and deletion of a region, if there is a bounded limit of memory which can be allocated, then the memory is allocated on the stack.
- Significant portion of inter region analysis is similar to implementations by Talpin, Birkedal, Tofte.

3rd Oct 2023

Work to be done

- Presentation of paper of Cornell - Thursday, Friday, Saturday, Sunday
- Reading of these papers:
    - Cornell - Extensive Reading
    - Free-Me
    - Conditional correlation analysis for safe region-based memory management
    - Safe and efficient hybrid memory management for Java

10th Oct 2023

- Presentation of Cornell paper is mostly done, need to iron out some inconsistencies discussed during it and finish up with Cornell paper.

- Reading of Free-Me paper in completion - 8th Oct
- Creating Presentation - 9th Oct
- Presentation of Free-Me paper - 10th Oct

17th Oct 2023

- Final Search for Papers:
- Other Important Papers:
    - IITB Heap Reference
    - Madrid Spain Heap Reference
    - Flow Control UTAustin
    - Data-Structure Information
- Final Papers of Relavance:
    - Compile-Time Deallocation of Individual Objects
    - Uniqueness Inference for Compile-Time Object Deallocation
    - CLOSER

31 Oct 2023

- Compile-Time Deallocation of Individual Objects
    - Thought of manipulating the necessary bounds to get better results
    - But program is based on reference counts of each object.
    - Not related
- Uniqueness Inference for Compile-Time Object Deallocation
    - Based on the similar principle of reference count of the object.
    - Only deals with checking if an object has only one reference and has object deletion methods in place when the unique reference is overwritten
    - Primary outcome of the above: Recursive deallocation of objects efficiently when parent node is overwritten causing entire heap structure to be efficiently removed.
- CLOSER *