Heap Liveness Analysis - A Dataflow Approach

Problem Statement

Memory management is one of the most important aspects to keep in mind when developing software. If a program uses too much memory or leaks memory, it is an undesirable effect and can lead to program crash too. In this project, We implement a compile-time liveness-based heap analysis approach to detect dead heap objects at every program point, and use this analysis pass to free up objects as soon as they become dead.

1 Liveness Based heap analysis

1.1 Memory Graph analysis

A memory graph is a directed graph which represents the heap memory and it's points-to and pointee information for a program. Our memory graph analysis is a linear iteration over the statements of a function and thus captures all the direct as well as indirect paths to a node in the graph.

However, it cannot deal with allocations in a loop or circular structures.

1.2 Access Path analysis

Our liveness-based heap analysis approach [2] is based on Dataflow analysis technique which is a popular framework for compile-time analysis of programs. We draw our motivation from a key insight that any data in the heap is accessed through the use of special expressions called access expressions [1]. e.g

$$a = b->f1->f2->f3$$

Here, $b->f1->f2->f3$ is an access expression. We will denote an access expression by α from now on. We define the following terminologies in relation the access expressions

- **Access Path** - Access Paths are denoted by ρ and are simply an abstract representation of access expressions, they're also of the form $a->f1->f2->f3$, where $f1$, $f2$, $f3$ are called the **links** in the access path. The $frontier(\rho)$ of the access path is the link $f3$.
- $base(\rho)$ is defined as the base of an access path, which is the access path, excluding it's frontier. Thus the $base(\rho)$ for the above access path is $a->f1->f2$.

Thus, if we can determine the access paths which are live after every program point, then this information can be correlated to the heap objects live after every program point. Consider the below assignment statement

$$x->r->n = y->n->n$$

For the above case, we consider the access paths that are genreated, killed and transfered by the above assignment

- Killed access paths - The above assignment kills all the access paths $x->r->n$ as a prefix
- Generated Access Paths - The access paths $x->r$ and $y->n$ $y->n->n$ are directly used by the expression and are thus generated. Thus all the $prefixes(x->r)$ and $prefixes(y->n->n)$ are generated by the statement
- Transferred access paths - Since the assignment means that both $x->r->n$ and $y->n->n$ point to the same heap location after the assignment, thus all the access paths of the form $x->r->n->\sigma$ where σ is a non-empty access path, are transferred to entry of program point as $y->n->n->\sigma$.

Statement s	$LKill_s$	$LDirect_s$	$LTransfer_s$
$\alpha_x = \alpha_y$	$\{\rho_x \rightarrow *\}$	$pref(base(\rho_x)) \cup pref(\rho_y)$	$\{\rho_y \rightarrow \sigma \mid \rho_x \rightarrow \sigma \in X\}$
$\alpha_x = new$	$\{\rho_x \rightarrow *\}$	$pref(base(\rho_x))$	ϕ
$Use\ \alpha_y$	ϕ	$pref(\rho_y)$	ϕ
$return\ \alpha_y$	ϕ	$pref(base(\rho_y))$	ϕ

Since, we are determining the liveness of access paths, this is a Backwards DFA. The statement-level transfer function for the DFA is

$$IN_S(X) = (X - LKill_s) \cup LDirect_s \cup LTransfer_s(X)$$

The meet operator for the DFA is

$$OUT_s = \cup(IN_s)$$

2 Implementation Details

We implement the above analysis as LLVM passes [3]. The dead object free flow is implemented in three steps across two LLVM passes. These three Steps are

- Step one - Computing the memory graph of a function. This is done in a seperate pass.
- Step two - Computing the In and Out sets of live Access Paths at every program point.
- Step three - This passes uses the memory graph pass and frees all the dead heap objects after every program point.

Step two and three are performed in the same LLVM Pass.

2.1 Intraprocedural Analysis

2.1.1 Memory Graph Pass

This is an analysis pass which calculates the memory graph for a function. The graph is represented by an adjacency matrix, with nodes being one of memory nodes (nodes which represent heap memory), field nodes(nodes which represent a field of a struct) or root node (nodes which represent the actual pointers in the program). The edges in the graph can be between any two nodes and can have either field names or dereference operator (*) as a label.

This pass analyzes each instruction sequentially and updates the memory graph as needed.

Four kinds of LLVM IR instructions are considered:

1. Function Calls - Function calls of the form

   ```
   %a = call i32* @foo(i32 %4)
   ```

   add a memory node in the graph, since they return a pointer (it is assumed that the pointer returned is generated by a malloc or similar calls). It also generates a root node for *%a* and adds an edge between *%a* and the memory node.

2. Casting instruction - We consider casting instructions which convert from one pointer type to another. These instructions are of the form

   ```
   %0 = bitcast i8* %call to i32*
   ```

   They generate a root node for %0 and also adds an edge between %0 and the memory node pointed to by *%call*.

3. Struct's field accessing instruction - These instructions are used to access the fields of a struct and are of the form (broken down into two lines for readability)

   ```
   [breaklines=true]
   %f33 = getelementptr inbounds %struct.A,
   %struct.A* %0, i32 0, i32 2
   ```

   . Here the accessed field is $\%0->2$ or the third field of the struct. This adds a field node to the graph and an edge from root node of %0 2 and this field node. It also generates a root node for $\%f33$ and adds an edge between this root node and field node.

4. Load Instruction - Load instruction of the form

   ```
   %3 = load %struct.B*, %struct.B** %f33, align 8
   ```

   also update the memory graph because they're loading a pointer into a register. The above load instruction generates a root node for %3 and adds an edge between this root node and the memory node pointed to by $dereference(\%f3)$.

5. Store Instruction - Store instruction of the form

   ```
   store %struct.C* %2, %struct.C** %f2, align 8
   ```

   store a pointer into the location pointed to by $\%f2$. Thus, the above statement gnerates a dereference edge between memory node pointed to by $\%f2$ and memory node pointed to by %2.

2.1.2 Access Path DFA

Since the Access Path DFA analysis is a non-seperable DFA, instead of operating on basic blocks and calculating their *IN* and *OUT* values, we operate on an instruction level, this also allows us to free heap memory as soon as it becomes dead. We consider these instructions from the LLVM IR to make changes to the access path set.

1. Function Calls - Function calls of the form

   ```
   %a = call i32* @foo(i32 %4)
   ```

   This is considered as a call to "new" instruction (i.e memory allocating instruction) with $\alpha_x = \%a$ and $\alpha_y = \phi$

2. Casting instruction - We consider casting instructions which convert from one pointer type to another. These instructions are of the form

   ```
   %0 = bitcast i8* %call to i32*
   ```

   This is an assignment instruction with $\alpha_x = \%0$ and $\alpha_y = \%call$

3. Struct's field accessing instruction - These instructions are used to access the fields of a struct and are of the form (broken down into two lines to improve readability)

```
[breaklines=true]
%f33 = getelementptr inbounds %struct.A,
%struct.A* %0, i32 0, i32 2
```

This instruction is assignment instruction with $\alpha_x = \%f33$ and $\alpha_y = \%0 \rightarrow 2$

4. Load instruction - Load instructions have two use cases depending upon the type of it's operands, which are:

   - Pointer load instruction - These are of the form

     ```
     %3 = load %struct.B*, %struct.B** %f33, align 8
     ```

     which loads the pointer value stored in $\%f33 \rightarrow *$ in %3, thus it is also an assignment instruction with $\alpha_x = \%3$ and $\alpha_y = \%f33 \rightarrow *$
   - Value load instruction - These are of the form

     ```
     %3 = load i32, i32* %f33, align 8
     ```

     This is a use type instruction with $\alpha_x = \%f33 \rightarrow *$

5. Store instruction - Store instructions also have two use cases depending upon the type of it's operands, which are:

   - Pointer store instruction - These are of the form

     ```
     store %struct.C* %2, %struct.C** %f2, align 8
     ```

     which stores the pointer value stored in %2 in $\%f2 \rightarrow *$, thus it is also an assignment instruction with $\alpha_x = \%f3$ and $\alpha_y = \%f33 \rightarrow *$
   - Value store instruction - These are of the form

     ```
     store i32 %2, i32* %f2, align 8
     ```

     This is a use type instruction with $\alpha_x = \%f33 \rightarrow *$

6. Return instruction - Return instructions which return a pointer are of the form

   ```
   ret i32* %0
   ```

   which are simply use instructions with $\alpha_x = \%0$

2.1.3 Dead Memory Freeing Pass

This pass is the one which modiefies the target IR and inserts calls to free at appropriate points. It uses the above two described passes as a dependency and calculates the memory graph and the Access Paths for the function being operated. It then iterates over every instruction in the function, calculates the memory nodes that are accessed by each access path and adds them to a list called *accessedNodes*.

It uses the following algorithm to then free memory after every instruction

Algorithm 1 The heap freeing algorithm

```
accessedNodes ← getAccessedNodes(Instruction)
memoryNodes ← getMemoryNodes(MemoryGraph)
for node ∈ memoryNodes do
    if node ∉ accessedNodes then
        freeInst ← free(node)
        insertAfter(Instruction, freeInst)
```

2.2 Interprocedural analysis

To extend our transformation pass to deal with interprocedural analysis, we mainly focus on two key aspects of interprocedural flow.

- Global pointer variables - Global pointer variables are not freed in any function except of the "main" function of the program. And in the main function of the program, the global variables are freed at the end. However, if a global pointer is not allocated in the main function, it cannot be freed since it can change the program semantics (as global variables are shared across functions).

- Functions returning pointers - Functions which return pointers, if they allocate memory for the pointer, then such pointers are not freed by our analysis and only freed in the caller function.

Our approach is context insensitive and currently cannot handle pointer operands to a function.

3 Evaluation

We evaluate our Dead objects freeing pass on two parameters

1. Correctness - The inserted free calls should not change the semantics of the original program

2. Accuracy - Ideally, all the memory allocated should be freed by our implementation.

We implement three microbenchmark to test our implementation

- *intraprocedural-linked-list-benchmark* - A simple linked list implementation

- *correctness-check-benchmark* - A benchmark with calls to free to test for double-free and use-after-free bugs.

- *interprocedural-benchmark* - A benchmark to test interprocedural transformation

3.1 Correctness

To evaluate correctness we imply that the free calls inserted can only change the semantics of program by two ways

1. Freeing object prior to the last use

2. Freeing the same object twice

We do not consider memory leak as a correctness issue since it does not change the semantics of the program itself.

To perform the correctness check, we use valgrind, which is a profiler and debugger for linux programs. Valgrind has a memcheck utility which acts like an address sanitizer and detects memory corruptions. To perform the check, we run

```
valgrind --tool=memcheck --leak-check=full benchmark
```

Below is the result of running valgrind with memcheck for our benchmarks

Benchmarks	Use-after-free check	Double free check
intraprocedural-linked-list-benchmark	✓	✓
correctness-check-benchmark	✓	✓
interprocedural-benchmark	✓	✓

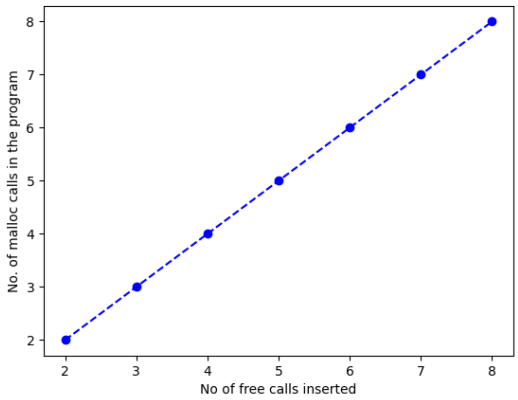
Thus, our transformation does not introduce any bug in the program and maintains the semantics of the program.

3.2 Accuracy

We test our transformation pass against two kinds of benchmarks, one which only requires intraprocedural transformation, the other for interprocedural transformation

For intraprocedural evaluation, we create linked lists in the target of varying length, we then record the number of malloc calls in the program as well as the number of free calls inserted by our transformation. For the sake of evaluation, we do not free any memory in the target source code, thus any memory freed is due to the transformation. The result is plotted as a graph below

Figure 1: No. of Free calls for intraprocedural analysis



3.3 Comparison against other implementations

We tried to find other compile time memory-freeing implementations but failed to find any existing widely used tool based on LLVM. An implementation of reference counting based approach exists but those were also JVM and swift based.

We found one open-source implementation (coincidently implemented for the same CS5544 course) called StaticHeap-Analyzer on github and compared our work against it. However, their transform passes failed to run which we tried debug but could not get it to work and we could not compare our work against them.

However, they mention that their implementation is specifically for intraprocedural analysis and cannot handle any interprocedural cases which we currently do.

4 Future Work

Our implementation can be extended to support two major features

1. Dealing with loops - Loops present a unique challenge, which is that they can give rise to infinitely long access paths, to deal with this problem, our access path DFA can be extended to use access graphs which are similar to Finite state machines and can have self loops.

2. Extending to full-fledged interprocedural analysis - Our implementation deals with a small subset of interprocedural tranformation, namely global variables and function calls (without pointer variables), however, a complete context-sensitive interprocedural analysis is also possible with the access graph approach.

5 Conclusion

Thus, in this project we implemented a liveness-based heap analysis and transformation tool which automatically frees up memory. Our implementation targeted a subset of programs and was successful in dealing with intraprocedural as well as some interprocedural transformation. We also evaluated our work on correctness and accuracy and established that our implementation performed well on both aspects.

References

[1] KHEDKER, U. P., AND KANVAR, V. Generalizing the liveness based points-to analysis. *CoRR abs/1411.5289* (2014).

[2] KHEDKER, U. P., SANYAL, A., AND SATHE, B. *Data Flow Analysis - Theory and Practice*. CRC Press, 2009.

[3] LATTNER, C., AND ADVE, V. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (2004), pp. 75–86.