CS6843 Jan-May 2016 Assignment-2 Specifications

Problem Statement: Write an LLVM pass to perform pointer and escape analyses.

Introduction

Dynamic memory allocation instructions, such as *malloc,* allocate memory on the heap and return the starting address, which may be stored into a pointer variable (*p* in the example below).

```
int *p = malloc(sizeof(int));          // r1
```

To release the memory, languages such as C need an explicit *free* instruction. Whereas in languages that support automatic garbage collection (such as Java), releasing allocated memory is done by the runtime system periodically, consuming precious time.

Points-to analysis helps in identifying the possible run-time memory locations that could be pointed-to by a variable. Thus, *p* points-to the memory allocated at line number *r1* after executing the above instruction.

For a function *f*, **escape analysis** helps in identifying the memory locations allocated within *f* that cannot be accessed outside *f*. Once we know the set of such locations, we can allocate them on the stack-frame of the function, instead of the global heap. In C, this can be done by replacing the *malloc* instruction with an *alloca* instruction. We do not need to explicitly release (or garbage collect) the memory allocated on the stack, as it will be released automatically when the lifetime of the allocating function is over (when the function returns).

In C, a memory location might escape to another function if:

(a) it is pointed-to by a variable that is passed as an argument to another function, or
(b) it is pointed-to by a global variable, or
(c) it is *reachable* by an already escaping memory location as defined in (a) and (b) above.

We define *reachability* based on a ***points-to escape graph***, which is constructed as follows:

Nodes:

(a) All variables that can point-to the heap
(b) All malloc statements

Edges:

1) malloc statement, *p* = *malloc*: Add a directed edge from the node representing *p* (create if does not exist) to the node representing *malloc* on this line. We say that *p* points-to the *malloc* node at this program point.
2) copy statement, *p* = *q*: Add edges from the node representing *p* to the nodes pointed-to by *q*.

3) load statement, *p = *q*: Add edges from the node representing *p* to the nodes pointed-to by the pointees of *q*.
4) store statement, **p = q*: Add edges from the nodes pointed-to by *p* to the nodes pointed-to by *q*.
5) function call, *foo(p,q)*: Mark all the *malloc* nodes reachable from *p* and *q* as escaping from the current function.
6) Ignore other statements. (Note: We will not have address-of operator (&) in this assignment.)

Evidently, the above analysis is intraprocedural, flow-insensitive, and field-insensitive. In practice, higher precision is desired; but we will keep it simple for the purpose of the assignment. Also, we would not have global variables for simplicity.
A points-to escape graph can also be used to obtain the points-to sets of objects, along with getting the escape information.

Input Specification

For this assignment, you only need to take care of the following types of statements:
mallocs, assignments (copy, load, store), function-calls.
Thus, there would be no structures, loops, if-else statements, etc.

Output Specification

We expect the output in the following format:

function-name: *list of space-separated node ids (temporary names representing malloc statements in LLVM IR) that escape the function*
temporary_n: *list of mallocs pointed-to by temporary_n*

For example,
foo: %2 %15
%3: %4 %7
%8: %2 %15
bar: %3 %16 %27 %33
and so on.

NOTE: We would specify the exact format of printing temporaries (whether a '%' sign is needed or not, etc.) within a couple of days.

Evaluation

For each testcase, 0.5 marks would be awarded for printing the escape information correctly, and 0.5 marks for printing the points-to information correctly. There will be 7 public testcases (released along with the specification), and 8 private testcases. We will upload a sample evaluation script shortly.

Further Reading (optional)

In general, escape analysis is more beneficial for object-oriented languages where garbage collection is done automatically. Further, it can also be used to identify thread-local objects, so that we can remove synchronization around the accesses to those objects, if any.
Following are some of the pointers for further reading:

1. Yair Sade. *Optimizing C Multithreaded Memory Management using Thread-Local Storage*. Thesis, School of Computer Science, Tel-Aviv University, Israel, 2004.
2. John Whaley and Martin Rinard. *Compositional Pointer and Escape Analysis for Java Programs*. OOPSLA 1999.
3. https://en.wikipedia.org/wiki/Escape_analysis