Lab Work – Alias Sets[1]

Name: _____ ID: _____

Alias Analysis is a code analysis technique that says which pointers may point to the same memory location. There is a vast literature about it. Wikipedia also provides a quick introduction on the subject[2].

We have made available an interprocedural LLVM pass that finds all the pairs of aliases between pointers. This pass also groups pointers in common sets. Two pointers, p_1 and p_2 are in the same set if one of these two conditions apply:

- p_1 and p_2 alias each other;
- There exists a pointer p, such that p and p_1 are aliases, and p and p_2 are also aliases.

We call these sets *alias sets*. The goal of this lab is to understand a bit about how LLVM uses alias analyses, and how to work on a module pass. Additionally, we will be discussing a bit on the difficulties of implementing precise and efficient pointer analyses.

1. Download and compile the `AliasSet.zip`[3] file, in the lab folder.

2. Try running it in the file below (`file1.c` in the lab folder):

```
int main() {
  int a[5] = {2, 3, 5, 7, 11};
  int b[5] = {13, 17, 19, 23, 29};
  int sum_a;
  int sum_b;
  int ia, ib;

  for (ia = 0; ia < 5; ia++) {
    sum_a += a[ia];
  }

  for (ib = 0; ib < 5; ib++) {
    sum_b += b[ib];
  }

  return sum_a + sum_b;
}
```

To run the pass, use the sequence of commands below:

```
$> clang -c -emit-llvm file1.c -o file.bc
$> opt -mem2reg -instnamer file.bc -o file.rbc
$> opt -load AliasSet.dylib -alias-set -print-as -pairwise file.rbc -disable-output
```

[1] The material necessary for this assignment is available at `http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/lab/exercises/AliasSet.tgz`

[2] `http://en.wikipedia.org/wiki/Alias_analysis`

[3] Credits for this implementation go to Victor Hugo Sperle Campos.

(a) How many pairs of aliasing pointers did you discover?

(b) How many alias sets?

(c) Our analysis end up indicating that pointers `a` and `b` alias. Is this aliasing possible during an actual execution of the program?

3. Try running the pass again, but this time, with a slightly different command line:

```
$> opt -load AliasSet.dylib -basicaa -alias-set -print-as -pairwise
file.rbc -disable-output
```

Take a look into `opt`'s documentation. What does the `basicaa` tag do?

4. Let's now consider this new file, which can be downloaded as `file2.c` in our folder.

```
#include <stdlib.h>

int* maxPointer(int* a, int* b) {
  return *a > *b ? a : b;
}

void swap(int* x, int* y) {
  int tmp = *x;
  *x = *y;
  *y = *x;
}

int main() {
  int* p0 = (int*)malloc(4);
  int* p1 = (int*)malloc(4);
  int* p2 = (int*)malloc(4);
  int* p3 = (int*)malloc(4);
  int* p4;

  *p0 = 0;
  *p1 = 1;
  *p2 = 2;
  *p3 = 3;
```

```
  p4 = maxPointer(p0, p1);
  swap(p2, p3);

  return *p0 + *p1 + *p2 + *p3 + *p4;
}
```

Compile and run this program. You can perform these steps via the commands below:

```
$> clang34 -c -emit-llvm file2.c -o file.bc
$> opt -mem2reg -instnamer file.bc -o file.rbc
$> opt -load AliasSet.dylib -basicaa  -alias-set -print-as file.rbc -disable-output
```

(a) How many alias sets do you obtain?

(b) You will notice that our analysis has set function arguments, e.g., `x` and `y` in `swap` as alias. The same has happened to `a` and `b` in `maxPointer`. This aliasing is clearly not necessary for this program. However, LLVM still must enforce it. Why does the compiler need to be so conservative?

(c) In your opinion, should pointers `main.p0` and `maxPointer.a` alias? What is the answer of our pass in this case?

5. So far, our alias set pass is intraprocedural. An intraprocedural analysis is confined to the bounds of functions. In other words, it does not see the program as a whole, but works on each individual function independently.

   (a) An intraprocedural analysis may have one big advantage over an equivalent interprocedural implementation. Which advantage that could be?

   (b) Similarly, an interprocedural analysis also has advantages over their intraprocedural counterparts. Explain one of these advantages.

(c) If we were to make our pass interprocedural, how do you think we would have to modify it?

6. Let's then implement the modifications that you have suggested in the last question. In the `runOnModule` function, whose implementation is available in the file `AliasSet.cpp`, you will find two `TODO` tags. Implement these missing parts of our pass:

   - In the first `TODO`, you must find a way to match actual and formal arguments of function calls. The formal arguments of a function are given by the names of the parameters that have been declared in the function. The actual arguments are the parameters passed to the function when it is called.
   - The formal return value is the name of the return value of the function. The actual return location is the variable that receives the value returned by the function.

   A few hints follow below, but the most important is this one: there exist already several passes in the LLVM distribution that implement similar matching.

   - In our pass, the variable `call` is a pointer to the instruction that represents the invocation site of the function.
   - The variable `call` itself is the actual return location of a function.
   - Take a look into how pointers are paired up to indicate that they are aliases. This pairing happens, for instance, in the intraprocedural part of the pass, via the call `sets->unionfind(p1, p2)`.
   - To go over the formal arguments of a function, take a look into the `CallSite::arg_iterator` data-type, available in the LLVM API.
   - The formal return values of the function have already been collected in our pass. Look for dynamic casts against `ReturnInst`.

   If you implement this matching correctly, then you should obtain only one set of aliases for `file2.c`:

```
$> clang34 -c -emit-llvm file2.c -o file.bc
$> opt -mem2reg -instnamer file.bc -o file.rbc
$> opt -load AliasSet.dylib -basicaa -alias-set -print-as file.rbc -disable-output

Number of alias sets: 1
swap.x main.call1 main.tmp swap.y main.call4 main.call maxPointer.a maxPointer.b
main.tmp3 main.call3 main.tmp2 main.call2 maxPointer.cond main.tmp1
```

7. In addition to `basicaa`, LLVM provides a few other implementations of alias analysis. The implementations available in clang 3.4 are listed below. Explain what each of these analyses does:

   - `scev-aa`:
   - `globalsmodref-aa`:
   - `libcall-aa`:
   - `tbaa`: