DCC888 – Pointer Analysis

Nome: _____ Matrícula: _____

1. Let p be a program variable, and let $[\![p]\!]$ be the set of locations pointed by p. Given this definition, we write the points-to analysis problem as a set of constraints, which have the following form:

$id = \texttt{malloc}_i$	$\{\texttt{malloc-i}\} \subseteq [\![id]\!]$
$id_1 = \&id_2$	$\{\&id_2\} \subseteq [\![id_1]\!]$
$id_1 = id_2$	$[\![id_2]\!] \subseteq [\![id_1]\!]$
$id_1 = *id_2$	$\&id \in [\![id_2]\!] \Rightarrow [\![id]\!] \subseteq [\![id_1]\!]$
$*id_1 = id_2$	$\&id \in [\![id_1]\!] \Rightarrow [\![id_2]\!] \subseteq [\![id]\!]$

   (a) Which constraints will be produced for the program below?

```
p = malloc;
x = y;
x = z;
*p = z;
p = q;
q = &y;
x = *p;
p = &z;
```

   (b) Find a valid solution for the constraint system that you have produced above.

2. Reaching definitions is a dataflow analysis that uses the following equations to describe the assignment $v = E$ at a program point p, where E is any right-hand side expression:

$$IN(p) = \bigcup OUT(p_s), p_s \in pred(p)$$

$$OUT(p) = (IN(p) \setminus \{defs(v)\}) \cup \{p\}$$

   How would you write equations for the store $*a = E$? Assume that the points-to set associated with variable a is given by $[\![a]\!]$.

3. Below we have a system of constraints produced out of a programming languages that has pointers:

```
h = &c          e = &g          b = c
h = &g          h = a           c = b
a = &e          f = d           b = a
d = *h          *e = f          f = &a
```

(a) Draw the constraint graph that represents the statements above. Your graph should represent the constraint system right before we start running the analysis. In this case, we assume that you shall use an Andersen-style analysis to solve this points-to problem.

(b) Draw the constraint graph at the end of the analysis, after all the candidate edges have already been evaluated. It is not necessary to collapse cycles.

(c) Write down a table that related variables with the locations to where these variables can point to. You discover these locations once you finish running the points-to analysis on the program.

4. This exercise refers to the language below. This assembly-like programming language has a special instruction $\texttt{out}(v)$, which prints out the contents of variable v in the standard output. The other instructions have obvious semantics. Instruction $\texttt{bzr}(v, l_1, l_2)$ changes the program flow to l_1 if v is zero, or to l_2 otherwise.

(Variables)	::=	$\{v_1, v_2, \ldots\}$
(Data instructions)	::=	
– (Read address)	|	$v_1 = \&v_2$
– (Assignment)	|	$v_1 = v_2$
– (Store in memory)	|	$*v_0 = v_1$
– (Load from memory)	|	$v_1 = *v_0$
– (Data output)	|	$\texttt{out}(v)$
– (Conditional branch)	|	$\texttt{bzr}(v, l_1, l_2)$
– (Inconditional jump)	|	$\texttt{jmp}(l)$

There exists a software vulnerability called *address leak*. A program presents this vulnerability if an adversary can observe any address in the program after running said program. If you want to know more about this kind of vulnerability, do know that adversaries can circumvent a protection called *Address Space Layout Randomization* (ASLR) if they can read an address from the program.

(a) What is ASLR, and why the knowledge of a program address lets an adversary circumvent this software protection?

(b) Write a program in our assembly language that suffers from this vulnerability.

(c) Write a data-flow analysis that determines if a program might present an address leak. Your analysis must have data-flow equations for every kind of instruction, and needs to take into consideration the fact that two variables might point to the same memory location. You can assume the existence of a table Π, such that $\Pi(v)$ gives the set of memory regions pointed out by variable v.

(d) Define the lattice onto which your analysis works.

5. This question refers to these two programs below:

```
void kernel1(int src[], int *s, int *acc) {
  int i;
  for (i = 0; i < *s; i++) { *acc += src[i]; }
}
void kernel2(int src[restrict], int *restrict s, int *restrict acc) {
  int i;
  for (i = 0; i < *s; i++) { *acc += src[i]; }
}
```

Once we compile and run both programs using LLVM 3.8, the second version, e.g., `kernel2` is almost 10x faster than the first version:

```
$> clang -O3 -disable-inlining a.bc -o a.exe
$> ./a.exe
Syntax: ./a.exe <array_size> <max_elem> <kernel>
$> ./a.exe 100000000 10000 1 # Kernel 1
Time spent = 0.328373
$> ./a.exe 100000000 10000 2 # Kernel 2
Time spent = 0.045527
```

(a) What is the meaning of the `restrict` keyword in C?

(b) Which optimizations do you think are responsible for speedups so dramatic?

(c) Why does the restrictification of function arguments enable such optimizations? Why couldn't clang carry them out without the indication that the pointers do not dereference overlapping memory regions?

(d) Can the compiler insert the `restrict` keywords in the pointer arguments automatically, so to transform `kernel1` into `kernel2` without the intervention of the user?