DCC888 - Static Analysis of Programs

Name: ____ **ID**: __________________

Name: ____ **ID**: __________________

By turning this exam in, I give my word that I have done it with my team only, on the understanding that we are allowed to consult any material publicly available, except material disclosed by other colleagues outside our team who are taking this course, or who have taken it in the past.

Goal: The goal of this exercise is to design support for a stochastic technique that predicts which load instructions are more likely to yield zero values. For instance, if `x = *v` is a load instruction, then we might say that it "yields" zeros with probability 0.83 if, whenever we run it, there is an 83% chance that the value assigned to `x` is zero. Predicting zeros is very important, because it helps us to design the so-called semi-ring optimizations (If you want to know more about it, see Leobas et al [1], but that's not necessary for the exam). Or, if we know that a load is very likely to yield zero, we can try to replace a dense matrix with a sparse implementation, for instance. But, of course, predicting the probability that a load yields zero is a very imprecise business. We can, at best, try to approximate the behavior of programs that we observe during execution.

Feature Vectors: To solve this exercise, we shall associate a feature vector with each load instruction in a program. A feature vector is an N-dimensional vector that holds together static data about the instruction. In this exercise, our feature vectors will have seven dimensions. So, consider the instruction below, that reads the memory in address `v`, and assigns the result into variable `x`. The instruction is located at program label `L`:

```
L: x = *v
```

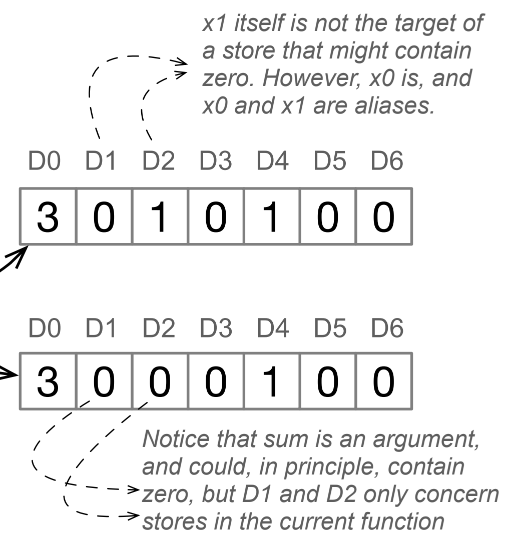
The following features will compose the feature vector. This is the vector formed by these features, represented as numbers. The order of the features is important. Thus, the first dimension of the feature vector is always D0, for instance:

- D0: M, where M is the number of times the constant zero appears in the program.
- D1: 1 if the current function contains a store of zero into the address of v; 0 otherwise.
- D2: 1 if the current function contains a store of zero into the address of an alias of v (v alias itself); 0 otherwise.
- D3: 1 if zero *must* be stored into the address of an alias of v; 0 otherwise.
- D4: N, where N is the number of loops that surround L.
- D5: 1 if the value of x *must* influence the result of a branch; 0 otherwise.
- D6: 1 if the value of x *might* be compared against zero; 0 otherwise.

Notice that D0 will be the same value for all the load instructions in a program. The other features depend on the individual load instruction. As an example, consider the program below, and its assembly representation on the right side.

```
01int* foo(int* sum, int N) {
02  int *aux = (int*)malloc(4*N);
03  for (int i = 0; i < N; i++) {
04    aux[i] = 0;
05  }
06  for (int j = 0; j < N; j++) {
07    aux[j] += sum[j];
08  }
09  return aux;
10}
```

```
    aux = malloc(N*4)
    i = 0
L02 p0 = i < N
    bz p0 L08
    x0 = &aux + i
    *x0 = 0
    i = i + 1
    jp L02
L08 j = 0
L09 p1 = j < N
    bz p1 L19
    x1 = &aux + j
    t0 = *x1
    x2 = &sum + j
    t1 = *x2
    t2 = t0 + t1
    *x1 = t2
    j = j + 1
    jp L09
L19 ret aux
```



x1 itself is not the target of a store that might contain zero. However, x0 is, and x0 and x1 are aliases.
D0 D1 D2 D3 D4 D5 D6
3 0 1 0 1 0 0
D0 D1 D2 D3 D4 D5 D6
3 0 0 0 1 0 0
Notice that sum is an argument, and could, in principle, contain zero, but D1 and D2 only concern stores in the current function

In the program above, we have two load instructions. Therefore, we extract two feature vectors. The feature vector is static: it will never change, and does not depend on the execution of the program. In the example above, the only difference is in D2, because the first load, which refers to array aux, might be assigned zero (through x0, which is an alias of x1).

How exactly we use a feature vector to carry out predictions is of no consequence to this question. However, just to give you a glimpse of how machine learning can be coupled with code analysis, imagine that if we have lots and lots of programs, we can extract lots and lots of feature vectors. Then, we can run these programs, and profile their execution. From this profile, we can associate each feature vector with the actual probability that the associated load produces a zero. And, from this data, formed by lots of feature vectors, plus the observed probabilities, we can build a regression model, that is, a function that, given a feature vector, outputs the probability that the associated load yields zero. For instance, below you can see a such predictor.

```
def predictor(D0, D1, D2, D3, D4, D5, D6):
    C0 = 1010
    C1 = -0.18
    C2 = 78.5
    C3 = -0.001
    C4 = 54.8
    C5 = -289630
    C6 = 0.75
    aux = C0*D0 + C1*D1 + C2*D2 + C3*D3 + C4*D4 + C5*D5 + C6*D6
    return 1 / (1 + math.exp(-aux))
```

Training the predictor consists in finding good values for the constants C0, C1, …, C6 that yield accurate predictions. We tune these constants observing known programs, and test them onto unknown programs.

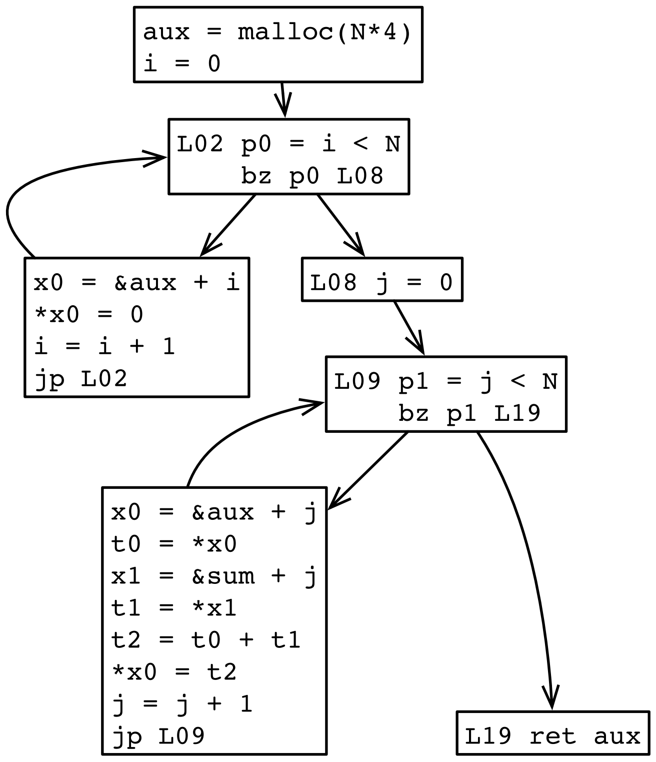
This is a linear predictor, but we could think about any kind of predictor, even using more complicated models like neural networks or random forests. For more about this modus operandi, see the work of Pereira et al [2].

What you must do. For each feature (and we have seven of them), you must design a static analysis that collects it. Thus, you might have to design seven static analyses. They might reuse common functions. The input for all of them is a sequence of binary instructions that represent the program in a low-level representation. This representation is not in SSA-form. It uses the following instructions:

Constant initialization	v = c, c in {…, -3, -2, -1, 0, 1, 2, 3, …}
Load	v = *u
Store	*u = v
Arithmetic operations	a = b ⊕ c, ⊕ in {+, -, *, /, <, >, <=, ==, !=, etc}
Branch if zero	bz v a, a is a program label
Unconditional jump	jp a, a is a program label
Memory allocation	a = malloc(c), c in {1, 2, …}

Q1 [2 Points]. Design an algorithm that converts the input program into a control flow graph. In other words, if you have a program like this one on the left side, your algorithm will produce the CFG on the right. On the other questions, we shall assume that you will be operating on the CFG.

```
    aux = malloc(N*4)
    i = 0
L02 p0 = i < N
    bz p0 L08
    x0 = &aux + i
    *x0 = 0
    i = i + 1
    jp L02
L08 j = 0
L09 p1 = j < N
    bz p1 L19
    x1 = &aux + j
    t0 = *x1
    x2 = &sum + j
    t1 = *x2
    t2 = t0 + t1
    *x1 = t2
    j = j + 1
    jp L09
L19 ret aux
```



Q2 [2 Points]. Design an algorithm based on a sliding-window (remember the notion of peephole optimization?) that computes D0 for the load instructions in the program.

Q3 [2 Points]. Design a static analysis to compute D1 for a load instruction L: x = *v. Notice that D1 is 1 if the constant zero might be stored into the address of v at some point in the program that reaches L; 0 otherwise. In other words, D1 is 1 if there is some instruction L': *v = y in the program, and y might contain the value zero and there is a path from L' to L. Notice that you will need some pre-analysis that will find out which variables might be zero in the program. In this way, you can know if variable 'y' above might hold the value zero before being stored into *v.

Q4 [3 Points]. Design a static analysis to compute D2 for a load instruction L: x = *v. D2 is very similar to D1; however, in this case, we must rely on alias analysis to find out if there is any alias of pointer v that might receive zero. You must not assume the existence of an alias analysis. You must design this alias analysis yourself.

Q5 [3 Points]. Design a static analysis to compute D3 for a load instruction L: x = *v. D3 also resembles D1, but there is an important difference. D1 is a "may" analysis, whereas D3 is a "must" analysis. Therefore, D3 is 1 if, and only if, regardless of the path the program takes, along this path, there is a store operation that deposits some value that is necessarily zero in an alias of v. For computing paths, try to think about dominance relations between labels.

Q6 [3 Points]. Design a static analysis to compute D4 for a load instruction L: x = *v. D4 holds the number of loops that surround the label L. You might assume that the target program contain only *natural loops*. However, you must find a way to count the number of natural loops that surround L. Call this number the *nesting level* of L. As an example, the load at L1 below has nesting level 1, and the load at L3 has nesting level 2:

```
L0: for (int i = 0; i < N; i++) {
L1:   *v = 0;
L2:   for (int j = 0; j < NN; j++) {
L3:     *z += w;
L4:   }
L5: }
```

Q7 [3 Points]. Design a static analysis to compute D5 for a load instruction L: x = *v. D5 is 1 if the value of x must influence the result of a branch. It is 0 otherwise. By "influence" we mean that the result of x can be used to compute the predicate of some branch. As an example, in the program below we have that x must influence the branch at L2, because every possible path in the program links L0 and L2. However, y might or might not influence the branch at L3, because there is a path that bypasses L3 after going over L1:

```
L0: x = *v
L1: y = *w
L2: if (x)
L3:   if (y)
L4:     *z += 1;
```

Q8 [2 Points]. Design a static analysis to compute D6 for a load instruction L: x = *v. D6 is 1 if the value of x might be compared against the constant zero; it is 0 otherwise. As an example, consider the program below. The variable x is compared against zero at L2, therefore, D6 of L0: x = *v is 1. However, y is not compared against zero, therefore, the D6 of "L1: y = *w" is zero.

```
L0: x = *v
L1: y = *w
L2: if (x < 0 && y > 1)
L3:     *z += 1;
```

References:

[1] Guilherme V. Leobas, Fernando Magno Quintão Pereira: Semiring optimizations: dynamic elision of expressions with identity and absorbing elements. Proc. ACM Program. Lang. 4(OOPSLA): 131:1-131:28 (2020)

[2] Fernando Magno Quintão Pereira, Guilherme V. Leobas, Abdoulaye Gamatié:
Static Prediction of Silent Stores. ACM Trans. Archit. Code Optim. 15(4): 44:1-44:26 (2019)