DCC888 – Sparse Data-Flow Analyses

Name: _____ ID: _____

1. Lets implement an analysis that finds the size of arrays. Our analysis will work on a very simple programming language, which has the API described in Figure 1(a). The analysis must associate an abstract state $[\![v]\!] = [t_l, t_h]$ to each variable v, where t_i is an element in the lattice seen in Figure 1(b). This lattice includes constants, and the integer variables that appear in the program. So, if we say that $[\![v]\!] = [t_l, t_h]$, we mean that the size of array v is not smaller than t_l, and no larger than t_h.

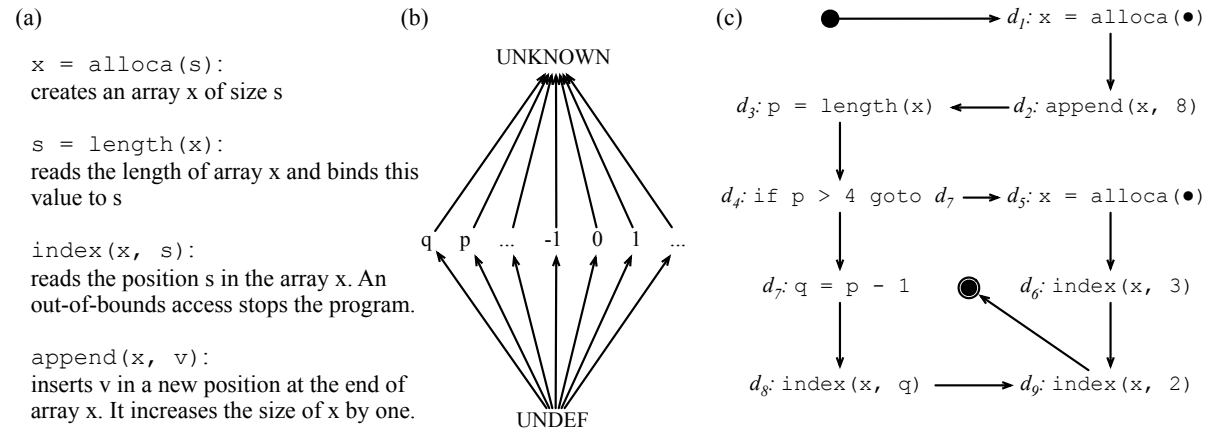

Figure 1: An instance of the array size inference problem.

(a) Which are the places where the analysis acquires information? Explain which information is learnt at each of these sites.

(b) How does the information propagates in this analysis? Does the information produced at each different site propagates in the same way?

(c) What is the meet operator of your analysis? Could you provide a table showing how it applies on the different elements of your lattice?

(d) Design a live range splitting strategy for this analysis.

(e) Show the program representation that you would produce for the control flow graph seen in Figure 1.

(f) Show the result of your analysis, after running it on the program representation that you have created in the previous question.

2. We still consider the array size inference analysis, but this time, lets assume that we know, for sure, that a program will never abort. In other words, if the program aborts due to an `index` statement, then this program is not a valid input to our analysis. How would this new assumption change your live range splitting strategy?

3. This question is about a data-flow analysis that estimates the bits of values stored in program variables [1]. The abstract state associated with a variable v is a list $[b_1, b_2, \ldots, b_n]$, where b_1 is the most significant bit of v, and b_n is the least significant bit. Each b_i can be either `0`, `1` or an unknown value `?`. Let's assume, in this question, a programming language with five different instructions: `assign`, `or`, `and`, `xor` and `not`. The semantics of each of these instructions should be obvious.

   (a) Try to infer the abstract states of each variable in the program below. We use `????` to denote an unknown initialization value. In this example, we assume that the maximum bitwidth of any variable is 4.

```
assign(v0, ????)  // v0 := ????
assign(v1, ????)  // v1 := ????
or(v2, v0, v1)    // v2 = v0 or v1
and(v3, v2, 0001) // v3 = v2 and 1
and(v4, v2, 0010) // v4 = v2 and 2
```

   (b) Let's define a forward analysis that assigns an abstract state to each variable in a program. We shall denote the abstract state of a variable v by $[\![v]\!]$ The transfer function for $\texttt{and}(t, u, v)$ is given below:

$$\frac{[\![u]\!] = [a_1, a_2, \ldots, a_n] \qquad [\![v]\!] = [b_1, b_2, \ldots, b_n]}{[\![t]\!] = [a_1 \ \mathit{and}\ b_1, a_2 \ \mathit{and}\ b_2, \ldots, a_n \ \mathit{and}\ b_n]}$$

   Define the transfer function for the other four instructions. Make a distinction between assignments between variables, and assignments of constants to variables.

[1] This analysis is, in fact, very useful in the synthesis of hardware. See the paper *Range and Bitmask Analysis for Hardware Optimization in High-Level Synthesis*, by Gort and Anderson

(c) Let's now add control flow to our programming language. To this end, let's assume two new instructions, `jump`(v, L), and `phi`$(v, v_1, \ldots, v_n)$. Define a *meet operator* $\wedge$ via a table, and write a transfer function to `phi` functions. What is the top of your analysis, given your definition of $\wedge$?

(d) We can learn information by adding a backward component to our analysis. Consider, for instance, the program below:

```
assign(v0, ????)
and(v1, v0, 0010)
```

We know that $[\![\texttt{v1}]\!] = 00?0 = ?0$. In other words, just the second least significant bit can vary. But, if `v0` is only used in the definition of `v1`, only its second least significant bit matters. Hence, we also know that $[\![\texttt{v0}]\!] = ??$. This information let's us narrow the range of bits that `v0` can assume. Define backward transfer functions to the five instructions of our original programming language: `assign`, `or`, `and`, `xor` and `not`.