Assignment 3
CS 704, Spring 2017
Due: May 4 2017

This assignment contains 2 questions; question 1 requires an implementation. Submit your assignment to aws-teach@cs.wisc.edu as a zip file containing the written part and the code.

1 \( k \)-induction

In class (and notes linked through website), we discussed \( k \)-induction as a technique for proving correctness of a program with respect to a post-condition. You are to implement the \( k \)-induction algorithm using the Z3 SMT solver. Your implementation should take a transition system as a formula, along with its initial states and post-condition. Given a value \( k \geq 1 \), your implementation should perform \( k \)-induction, returning “success” if \( k \)-induction succeeds, and “failure” otherwise.

- It is highly recommend that you use the Python API of the Z3 SMT solver (github.com/Z3Prover/z3). Z3 has other language bindings, but Python is the easiest to prototype with and will make you most productive.

- You are expected to exhaustively comment your code and describe what parts of \( k \)-induction you are implementing.

- You should encode three non-trivial transition systems and their respective post-conditions. All transition systems you provide should be correct with respect to their post-conditions. One of the transition systems should require \( k \geq 2 \) for the induction to succeed. Explain the programs your transition systems encode. For the example where \( k \geq 2 \) is required, explain why \( k = 1 \) is insufficient. It is OK for your examples to be hard-coded.

- Provide a transition system for which \( k \)-induction cannot prove correctness for any value of \( k \). Argue why that is the case.
2 Abstract interpretation and least fixpoints

This question concerns the correctness and efficiency of algorithms for abstract interpretation of flow-chart programs, as defined in the notes. For this question, you are encouraged to carefully revisit the notes provided on abstract interpretation.

For this question, assume that the nodes of the flow chart are numbered from 1 to \( k \), that each node has at most a constant number of predecessors and a constant number of successors, and that the abstract value for a given node comes from a finite-height domain \( D \), where the height of \( D \) is \( h \). The abstract functional \( f_a \) is thus of type \( D_k \to D_k \), and \( \text{fix } f_a \) can be expressed as

\[
\text{fix } f_a = \bigcup_{i=0}^{\infty} f^i_a(\bot_{D_k}).
\]

(\( f_a \) is the abstract interpreter we defined in class. Recall that we computed its least fixpoint by iteratively growing it from \( \bot_{D_k} \). Here, \( D_k \) is the abstract domain that defines the reachable states at the \( k \) program locations.)

Further assume that we can perform a join of two \( D \) values in constant time, and that we can also iterate over the successors of a flow-chart node in constant time.

Finally, assume that \( f_a \) has the property that when \( f_a \) is applied to a value \( x \in D_k \), the \( i \)th component of its output (which corresponds to flow-chart node \( i \)) depends only on the slots of \( x \) that correspond to node \( i \)'s predecessors in the flow chart. Let's denote the computation that computes the \( i \)th “part” of \( f_a(x) \) by \( f_{a,i}(x) \), and let's assume that an application \( f_{a,i}(x) \) can be performed in constant time. Let's now consider how an algorithm to determine \( \text{fix } f_a \) could be implemented as an iterative algorithm in an imperative programming language, such as C. One way would be to have two arrays of \( D \)'s, say \( \text{val}[1..k] \) and \( \text{temp}[1..k] \), and use the following procedure:

\[
\text{for } i = 1 \text{ to } k \text{ do } \quad \text{val}[i] = \bot_D
\]

\[
\text{while changes to any components of } \text{val} \text{ occur do }
\]

\[
\text{for } i = 1 \text{ to } k \text{ do } \quad \text{temp}[i] = f_{a,i}(\text{val})
\]

\[
\text{for } i = 1 \text{ to } k \text{ do } \quad \text{val}[i] = \text{temp}[i]
\]

Part 1 What is the asymptotic worst-case cost for computing \( \text{fix } f_a \) by the above procedure?

Part 2 The above algorithm is “round-robin iteration”. An alternative approach is to use the so-called “chaotic-iteration” algorithm:
for $i = 1$ to $k$ do
    $\text{val}[i] = \perp_D$
worklist = $\{i \mid 1 \leq i \leq k\}$
while worklist $\neq \emptyset$ do
    select and remove a node-index $i$ from worklist
    $\text{foo} = f_{a,i}(\text{val})$
    if $\text{foo} \neq \text{val}[i]$ then
        $\text{val}[i] = \text{foo}$
        for each successor $j$ of node $i$ do
            worklist = worklist $\cup \{j\}$
What is the asymptotic worst-case cost of the chaotic-iteration algorithm?

**Part 3** It is clear that the round-robin-iteration algorithm finds $\text{fix } f_a$, because it directly implements a climb up the sequence $[\perp_D, f_a(\perp_D), \ldots, f_a^i(\perp_D), \ldots]$. However, it is not so clear what the chaotic-iteration algorithm is doing: Because single slots of $\text{val}$ are treated individually on each iteration of the while loop, the chaotic-iteration algorithm is not implementing a climb up the Kleene sequence. The question then is: Is the chaotic-iteration algorithm a correct way of computing $\text{fix } f_a$?

For this part, show that it is a correct way of computing $\text{fix } f_a$ by arguing that the following properties hold:

1. The chaotic-iteration algorithm terminates.
2. The chaotic-iteration algorithm finds a fixed point.
3. The chaotic-iteration algorithm finds the least fixed point (namely $\text{fix } f_a$).