CS-226 Computability Theory, Coursework 1

Sean Handley, 320097@swan.ac.uk

March 2006

1

1.1 Show that, if A is finite, then $|\mathcal{P}(A)| = 2^{|A|}$

Suppose A = $\emptyset$

$\mathcal{P}(A) = \{\emptyset\}$ therefore $|\mathcal{P}(A)| = 1$

$|A| = 0$

$|\mathcal{P}(A)| = 2^0 = 1$

The base case holds.

Suppose $A = a \cup B, a \notin B$ therefore $|A| = |B| + 1$

$|\mathcal{P}(B)| = 2^{|B|}$ and $|\mathcal{P}(A)| = 2^{|B+1|}$

Therefore:

$\mathcal{P}(A) = \mathcal{P}(B)$ + induction step $(+1)$

$2^{|B+1|} = 2^{|(B+1)|}$

Therefore the induction step holds and $|\mathcal{P}(A)|$ is always equal to $2^{|A|}$ where A is finite.

1.2 Show that $|A| \neq |\mathcal{P}(A)|$.

If $|A| = |\mathcal{P}(A)|$ holds true for any value, then it contradicts the proof in 1.1, which is not intuitively feasible.

However, suppose $|A| = |\mathcal{P}(A)|$.

Let $A = \emptyset$.

$|\mathcal{P}(A)| = 2^{|A|} = 2^0 = 1$

But $|A| = 0$, a contradiction.

Therefore, we can conclude that $|A| \neq |\mathcal{P}(A)|$.

2 Define for sets A, B that $A \leq B$ if, and only if, there exists an injective function $f : A \to B$. Show that $\leq$ is reflexive and transitive. Is $\leq$ symmetric? Prove your answer.

Def: $A \leq B \iff f : A \to B$ s.t. $\forall a, f(a) \to f(b), a \in A, b \in B$

Reflexivity:

$A \leq B \sim A' \leq B'$

if, and only if, there exists an injective function

$f : A \to A'$ s.t. $\forall a, \forall a', f(a) \to f(a'), a \in A, a' \in A'$

and $f : B \to B'$ s.t. $\forall b, \forall b', f(b) \to f(b'), b \in B, b' \in B'$

Transitivity: $A \leq B, B \leq C \to A \leq C$

$f : A \to B$ s.t. $\forall a, f(a) \to f(b), a \in A, b \in B$

$f : B \to C$ s.t. $\forall b, f(b) \to f(c), b \in B, c \in C$

$f : A \to C$ s.t. $\forall a, f(a) \to f(c), a \in A, c \in C$

Symmetry: $A \leq B \sim B \leq A$

Suppose $|A| \leq |B| \sim |B| \leq |A|$

According to the above definition,

$A \leq B \iff f : A \to B$ s.t. $\forall a, f(a) \to f(b), a \in A, b \in B$

and $B \leq A \iff f : B \to A$ s.t. $\forall b, f(b) \to f(a), a \in A, b \in B$.

Assuming $\leq$ is indeed a symmetric relation, then it must also be a bijection. Therefore $\leq$ is always symmetric where $|A| = |B|$.

Now we must prove $|A| \leq |B| \sim |B| \leq |A|$ where $|A| \neq |B|$. Put more simply, $|A| < |B| \sim |B| < |A|$.

Proof by induction.

Let $|A| = 0$. This means $|B| > 0$. Let $|B| = 1$.

This holds true for $|A| < |B|$ but not for $|B| < |A|$, giving a contradiction. If we change our original conjecture to $|A| < |B| \not\sim |B| < |A|$ then the base case holds and we can move on to the induction step.

Suppose $A = a \cup B, a \notin B$ therefore $|A| = |B| + 1$

This means $|B| < |A|$ because $|A|$ is always bigger than $|B|$ by one. Consequently, $|A| < |B|$ can never happen. Therefore, $<$ is never symmetrical and $\leq$ is only symmetrical when A and B are equinumerous.

3 Which of the following sets are countable? Prove your answer.

A set is countable if it is finite or is equinumerous with $\mathbb{N}$.

3.1 The set $A := \{n \in \mathbb{N} | n \geq 5\}$.

$\mathbb{N}$ is countable, therefore $\mathbb{N} - \{1, 2, 3, 4, 5\}$ is also countable because $|\mathbb{N}| > |\mathbb{N} - \{1, 2, 3, 4, 5\}|$.

3.2 The set $\mathcal{P}(A)$ where A is as in 3.1.

$\mathcal{P}(A)$ is not countable because $\mathcal{P}(\mathbb{N})$ is not countable.

3.3 The set $B := \{n \in \mathbb{N} | \leq 5\}$.

This is countable because it is the set {1,2,3,4,5}.

3.4 The set $\mathcal{P}(B)$ where B is as in 3.3.

This is countable because it is finite.

3.5 The set of finite subsets of $\mathbb{Z}$.

This is not countable because the set of finite subsets of $\mathbb{Z}$ is not equinumerous with $\mathbb{N}$

3.6 $\mathcal{P}(\mathbb{Z})$.

This is not countable because it is not equinumerous with $\mathbb{N}$.

3.7 The set of infinite subsets of $\mathbb{Z}$.

This is not countable because it is not equinumerous with $\mathbb{Z}$.

3.8 The set $C := \{f | f : \mathbb{N} \to \{0, 1, 2\}\}$.

This set is countable because it is equinumerous with $\mathbb{N}$. This is provable because there exists a bijection $f : \mathbb{N} \to \{0, 1, 2\}$.

3.9 The set $D := \{(x_0, ..., x_{n-1}) \in \mathbb{N}^* | n \in \mathbb{N}, x_i \in \mathbb{N}, x_i \neq x_j,$ for $i \neq j\}$.

This is the set of all strings over $\mathbb{N}^*$ where each string is made up of unique natural numbers. It is countable.

4 Determine a URM program which computes the function $f : \mathbb{N} \to \mathbb{N}, f(x) := 2 \cdot x$. Justify your answer.

$I_0 = iszero(0, 5)$
$I_1 = succ(1)$
$I_2 = succ(1)$
$I_3 = pred(0)$
$I_4 = iszero(2, 0)$
$I_5 = iszero(1, 11)$
$I_6 = succ(2)$
$I_7 = succ(0)$
$I_8 = pred(1)$
$I_9 = iszero(1, 11)$
$I_{10} = iszero(3, 6)$

If $R_0 = 0$ then $2 \cdot x = 0$. Execution jumps to I_5 which jumps to I_{11} because R_1 is zero. This ends the execution.

If $R_0 > 0$ then I_0 returns $false$ and execution moves to I_1. I_1 and I_2 each increment R_1 by one. I_3 decrements R_0 by 1. At this point, I_4 checks if R_2 is zero, which it should be. This returns us to I_0 and the process loops

until $R_0 = 0$, leaving R_1 holding double the value that was initially in R_0. At this point, I_0 returns *true* and execution jumps to I_5, which returns $false$, allowing execution to flow to I_6. At this point, we need to copy the value of R_1 (which is now our answer) into R_0. I_6 increments R_2 by one, which acts as a flag, showing that the answer has been computed and is now to be moved into R_0. I_7 increments R_0 (which is now zero) by one. I_8 decrements R_1 by one. I_9 checks if R_1 is zero. If it is, then the answer has been moved to R_0 and execution jumps to I_{11}, which brings us to a halt. If R_1 is not zero, execution flows to I_{10} which checks if R_3 is zero. It will be because we haven't used it, therefore it returns *true* and executon continues from I_6 until R_0 holds the answer of the computation.

5 Determine the function $U^{(2)}$ computed by the following URM program U:

$I_0 = iszero(0, 4)$
$I_1 = pred(0)$
$I_2 = succ(1)$
$I_3 = iszero(2, 0)$
$I_4 = iszero(1, 8)$
$I_5 = pred(1)$
$I_6 = succ(0)$
$I_7 = iszero(2, 4)$

$U^{(2)} = f(a, b) \rightarrow a + b.$

5.1 Justify your answer. What happens if one omits $I_4 - I_7$ from the program?

If both starting inputs, R_0 and R_1, are zero, then I_0 returns *true* and execution jumps to I_4, which also returns *true*, jumping to I_8 and ending execution.

If $R_0 = 0$ and $R_1 > 0$, I_0 returns *true* and execution jumps to I_4 which returns $false$ and execution carries on to I_5, which decrements R_1 by one. I_6 increments R_0 by one. I_7 checks if R_2 is zero, which it isn't because it hasn't been used. Execution returns to I_4 and the loop continues until the value in R_1 is in R_0. At this point I_4 returns *true* and execution ends by jumping to I_8 leaving the value of $R_1 + 0$ in R_0.

If $R_0 > 0$ and $R_1 = 0$, I_0 returns $false$ and the answer is computed in the same way as before, only using instructions $I_0 - I_3$ until the answer is in R_1. The answer is then moved to R_0 as before.

The execution where $R_0 > 0$ and $R_1 > 0$ proceeds in a simlar fashion to the previous two cases.

If $I_4 - I_7$ are omitted, $U^{(2)} = f(a, b) \rightarrow 0$. This is because the value computed into R_1 is never moved back to R_0.

6 Write a Java program which simulates a URM.

```
import java.util.*;

/**
 * URM Simulator.
 *
 * @author Sean Handley, 320097@swan.ac.uk
 * @version March 2006
 */
public class URM {
        //container for registers/instructions
        private ArrayList registers, instructions;
        private int PC; //program counter

        //inner classes defining the 3 operations
        private class Succ {
                private int reg;
                public Succ(int reg) {
                        this.reg = reg;
                }
                public int getReg() {
                        return reg;
                }
        }

        private class Pred {
                private int reg;
                public Pred(int reg) {
                        this.reg = reg;
                }
                public int getReg() {
                        return reg;
                }
        }

        private class IsZero {
                private int reg, next;
                public IsZero(int reg, int next) {
```

```
                this.reg = reg;
                this.next = next;
        }
        public int getReg() {
                return reg;
        }
        public int getNext() {
                return next;
        }
}

/**
 * Initialises the simulator.
 */
public URM() {
        //ArrayLists allow an infinite number of elements
        //(up to the limit of the machine's memory)
        registers = new ArrayList();
        instructions = new ArrayList();
        PC = 0; //set the program counter to 0.
                        //This point to the current instruction.
}

//this is where instructions are specified
private void load() {
        //EXAMPLE FROM QUESTION 4.
        //Binary addition function, f(a,b) -> a + b

        //set the registers with our start values
        //e.g. a binary operation using the values 2 and 3

        //registers.add(Index, Object);
        registers.add(0,new Integer(2));
        registers.add(1,new Integer(5));

        //NOTE: The values must be integer OBJECTS rather
        //than primitives
        //(this is because we're using an ArrayList).
```

```
        //Instruction <identifier> = new Instruction(args);
        IsZero isZero1 = new IsZero(0,4);
        //all instructions must be added to the list
        instructions.add(isZero1);

        Pred pred1 = new Pred(0);
        instructions.add(pred1);

        Succ succ1 = new Succ(1);
        instructions.add(succ1);

        IsZero isZero2 = new IsZero(2,0);
        instructions.add(isZero2);

        IsZero isZero3 = new IsZero(1,8);
        instructions.add(isZero3);

        Pred pred2 = new Pred(1);
        instructions.add(pred2);

        Succ succ2 = new Succ(0);
        instructions.add(succ2);

        IsZero isZero4 = new IsZero(2,4);
        instructions.add(isZero4);

}

private void run() {
        Succ succ = null;
        Pred pred = null;
        IsZero isZero = null;
        int whichInst = 0; //represents the instruction

        System.out.println("Starting values:");

        for(int i = 0; i < registers.size(); i++) {
```

```
        System.out.println("R" + i + "=" +
         ((Integer)registers.get(i)).toString() + " ");
}

System.out.println();

for(PC = 0; PC < instructions.size();) {

        //determine which instruction we're executing
        try {
                pred = (Pred) instructions.get(PC);
                whichInst = 1;
        }
        catch(ClassCastException e) {
                try {
                        succ = (Succ) instructions.get(PC);
                        whichInst = 2;
                }
                catch(ClassCastException f) {
                        try {
                                isZero = (IsZero) instructions.get(PC
                                whichInst = 3;
                        }
                        catch(Exception g) {
                                System.out.println(
                                "Something's gone wrong. "
                                + "Check your URM instructions.");
                                System.exit(1);
                        }
                }
        }
        //now we have the instruction, let's get to work on it
        switch (whichInst) {
                case 1:
                        doPred(pred.getReg());
                        break;
                case 2:
                        doSucc(succ.getReg());
```

```
                                break;
                        case 3:
                                doIsZero(isZero.getReg(),isZero.getNext());
                                break;
                        default:
                                //do nothing
                }
        }
}

//fill null elements in the arraylist with zeroes when referred to
private void fillZeroes(int reg) {
        int theSize = registers.size();
        for(int i = theSize; i <= reg; i++) {
                registers.add(i,new Integer(0));
        }
}

private void doPred(int reg) {
        System.out.println("[Pred][Reg: " + reg + "]");
        fillZeroes(reg);
        if(((Integer)registers.get(reg)).intValue() > 0) {
                registers.set(reg,
                 new Integer(((Integer)registers.get(reg)).intValue() - 1));
        }
        System.out.println("R" + reg + " = " +
         ((Integer)registers.get(reg)).toString());
        System.out.println();
        PC++;
}

private void doSucc(int reg) {
        System.out.println("[Succ][Reg: " + reg + "]");
        fillZeroes(reg);
        registers.set(reg,
         new Integer(((Integer)registers.get(reg)).intValue() + 1));
        System.out.println("R" + reg + " = " +
         ((Integer)registers.get(reg)).toString());
```

```
                System.out.println();
                PC++;
        }

        private void doIsZero(int reg, int next) {
                System.out.println("[IsZero][Reg: " +
                 reg + "][Next: " + next + "]");
                fillZeroes(reg);
                boolean isZero = false;
                if(((Integer)registers.get(reg)).intValue() == 0) {
                        PC = next;
                        isZero = true;
                } else {
                        PC++;
                }
                System.out.println("(R" + reg + " = 0) = " +
                 isZero + ", next = " + PC);
                System.out.println();
        }

        private void outputResult() {
                System.out.println("R0 = " + registers.get(0));
        }

        public static void main(String args[]) {
                URM urm = new URM();
                urm.load();
                urm.run();
                urm.outputResult();
        }
}
```

7 The programming language Agda has a termination checker, which checks whether the given program is guaranteed to terminate. Prove that we cannot write a perfect termination checker which accepts all terminating programs.

This was originally proved by Alan Turing and employs diagonalisation and proof by contradiction. Informally, the proof is as follows.

> Suppose there is an algorithm, $halts(a, i)$, which takes an algorithm, a, and an input, i. If the algorithm halts on the given input, then $halts(a, i)$ returns $true$.
>
> We then construct an algorithm, $test(s)$, which takes a string, s, and runs the $halts$ algorithm as a subroutine, giving it s for both the a and i inputs. If $halts$ returns $true$ then $test$ returns $false$ else goes into an infinite loop.
>
> We then run $test(t)$ where t is the string representing the program $test$.
>
> If $test(t)$ halts, we can assume that $halts$ returned $false$, but this would imply that $test(t)$ should not have halted.
>
> If $test(t)$ does not halt, then it would imply that either $halts$ returned $true$, or that $halts$ itself went into an infinite loop. Therefore, $test(t)$ should have halted or $halt$ does not work for all inputs.

Turing's proof assumes the algorithms are implemented on a Turing Machine. The Church-Turing thesis proved that any computer which can feasibly be built is equivalent to a Turing Machine, thus all Turing Machines are Universal Machines.