Models of Computation
Revision Notes

James Brown

April 7, 2017

Contents

1 Introduction

These are notes I have written in preparation for the upcoming 2017 Models of Computation exam. This year the module was run by Paul Levy (P.B.Levy@cs.bham.ac.uk). This module is about problems and *computers*. We ask ourselves:

- What problems can be solved on a computer?

- What problems can be solved on a computer with finitely many states?

- What problems can be solved on a computer with only finitely many states, but also a stack of unlimited size?

- What problems can be solved on a computer with only finitely many states, but also a tape of unlimited size that it can read and write to?

- What problems can be solved *fast* on a computer?

- What does "fast" mean anyway?

- What does *computer* mean anyway?

2 Language Membership Problems and Regular Expressions

2.1 Language Membership Problems

Suppose we have a set of characters Σ, which we will call the *alphabet*. A *word* is a finite sequence of characters, and we write Σ^* for the set of all words. We can *concatenate* words. A *language* is a set of words and a subset of Σ^*. Given a word, we want to know is it in the language or not? If we take an example alphabet a, b, c, here are some languages:

- All words which contain exactly 3 b's

- All words whose length is prime

- All words that have more b's than a's

- The words abc, bac and cb

- No words at all

- The empty word

These examples are largely pretty useless, but this problem does have real world applications such as

- Java has rules about what you can call a variable. Is the word read by the compiler a valid variable name?

- A user makes an account and enters a password, is it valid?

- A student has submitted code for an assignment, is it correct?

- Will this code crash when it's run?

In each one of these examples, we are provided with a word and we want to know whether it is an acceptable word. We want to make a computer tell us the answer.

2.2 Regex

Regular Expressions are a useful notation for describing languages. We write Empty for the language consisting of no words (the empty set). We write a for the language consisting of just the single-character a, and ϵ for the language consisting of just the empty word. If we have a language L and L', we write LL' for the set of words that are a concatenation of a word in L and a word in L'. We can also write L—L' for the set of words that are in L or L' - the union of the two languages. We write L* for the set of words that are a concatenation of some number of words in L (some number may be 0). Just like arithmetic, regular expressions have precedence rules. $*$ has the highest precedence, then juxtaposition and then |.

Regular expressions in theoretical computer science mean an expression built from the above operations and nothing more. Regular expressions are used much more widely in programming with far more operations, but they cannot be used in the module. For example, + is a very common operation but not available to us.

3 Finite State Automata

4 Regular Languages

5 Bisimulation and Minimisation

6 The Halting Problem

In Computer Science there are many problems that we encounter. These all have an input (some string, a number, three strings etc.) and an output. Often this output is yes or no, but it may be a number or string or some other value. If a problem returns a yes or no result, then it is a decision problem. We have to draw a distinction between problems and instances. 'Is the word *abbab* accepted by the regexp $ab*(ab*|b)$?' is an instance of a more general problem 'Is a word w accepted by the regexp $ab*(ab*|b)$?'. Even more generally we could describe this problem as 'Given a word w and a regexp E, is w accepted by E?'.

Typically we want to consider problems which have a countably infinite set of acceptable inputs and acceptable outputs. Refer to Mathematical Techniques for Computer Science to see how we can ensure something is countably infinite. A problem can also be called a function, and we can also describe it as a subset of the set of inputs.

- Problem: given a positive integer, is it prime?

- The function that maps a positive integer to 'true' if it's prime and 'false' otherwise

- The subset of the set of positive integers consisting of all prime integers

If the set of inputs is the set of words over some alphabet, then the subset will be a language.

Key concepts:

- A decision problem (a problem which returns a yes or no answer) is decidable if it can be solved on a computer.

- A function is computable if it can be computed on a computer.

- A subset of the set of inputs (a language) is decidable if the corresponding decision problem can be solved on a computer.

We still haven't come up with a definition for *computer* yet! We get around this for now by saying a problem is 'Java-decidable' or 'Java-computable' for example, meaning that we can accomplish the task with a Java program.

6.1 Reducing a problem to another problem to solve the Halting problem

Suppose we have two problems - P and Q. If we can show how to solve Q using a black box which solves P then we can say that we have reduced the problem Q to the problem P. If we can reduce Q to P then if P is decidable, Q is decidable and if Q is undecidable, P is undecidable.

Given a nullary Java method such as the one below, can we tell if it will terminate or not? We keep things simple by assuming integers are unbounded, no exceptions are thrown and any such method when called either terminates or hangs forever).

```
void f (){
    ...
}
```

This is an example of the **Halting Problem**. Turing proved that the halting problem was undecidable. To prove this, we assume that it is decidable.

1. Consider the **unary halting problem**. Given a unary Java method

   ```
   void f (String x) {
       ...
   }
   ```

   and a string y, does `f` terminate when called with y? We reduce the unary halting problem to the nullary one. Given the unary method `f` and a string x, obtain a nullary method `g` by taking the code of `f` and replacing `x` with y - that is substitute in the argument values. Then, `g` terminates when called if `t` terminates when called with argument y. Since we assume the nullary halting problem is solvable, the unary one is too. This gives us a program

   ```
   boolean haltcheck (String somemethod, String y)
   ```

   where `somemethod` is the body of a unary method. When we apply with M and y this method returns true when M applied to y terminates, otherwise it returns false.

2. We build on this method further

   ```
   void hangcheck (String somemethod, String y) {
     if haltcheck(somemethod, y) {
        while true {}
     } else {
        return;
     }
   }
   ```

This method when applied to M and y, hangs if M applied to y terminates, otherwise it returns.

3. We build on this even further with a new program

```
void doublehang (String y) {
    haltcheck(y, y)
}
```

This method when applied to y (the body of the unary method), will hang if y applied to y terminates, otherwise it returns.

4. Finally, let z be the body of `doublehang`. We see that `doublehang`, when applied to z, terminates if and only if it hangs. This is a contradiction, so there cannot be any program which solves the halting problem.

7   Properties of Code

8   Turing Machines

9   Church's Thesis

10   Complexity and P

11   NP

12   Lambda-calculus

List of Figures

Index