Models of Computation
Revision Notes

James Brown

April 25, 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

7.1 Rice's Theorem

Rice's theorem states that every non-trivial semantic property of code is undecidable. We define a trivial property is one which either holds for everything or holds for nothing.

To prove Rice's theorem, we let α be a non-trivial semantic property of code and let `alwayshang` be a piece of code that always hangs regardless of argument. We say that `alwayshang` doesn't satisfy α, and since α is non-trivial we can say that there is another piece of code C that does. For any nullary program P, we form $F(P)$ by inserting P at the start of the code C.

- If P terminates, then $F(P)$ has the same semantics as C. Because α is semantic and C satisfies α, $F(P)$ does too.
- If P hangs, then $F(P)$ has the same semantics as `alwayshang`. Since α is semantic and `alwayshang` doesn't satisfy α, $F(P)$ also doesn't.

In short, P will terminate if and only if $F(P)$ satisfies α. This means that if we can test for α, we can also test whether P terminates by testing if $F(P)$ satisfies α. We've already proven that we can't prove P terminates, so we must not be able to test for α - meaning α is undecidable.

8 Turing Machines

In 1936, Alan Turing invented the **turing machine**. Turing machines are very simple computers, and have a very limited set of instructions that they can execute. They have finitely many states, but also have an infinite tape which they can compute upon. Because of this, a Turing machine is an idealized computer and can never exist in the physical world. Turing machines are very convenient for analysing the running times of algorithms. Firstly, the time taken for an algorithm scales straightforwardly as the size of the input increases. They typically tend to treat inputs of size ten or one million in the same way. Secondly, every step of computation is made explicit in a very conservative way.

Turing machines have finitely many **states** and an infinite tape, with a **head** which sits over one space of the tape. They also have a finite tape alphabet T, and each space

on the tape contains a letter which is contained in T. T also contains the blank character, $\textvisiblespace \in T$, which is particularly important. All but finitely many spaces on the tape are blank - it never happens that infinitely many spaces contain actual, non-blank, characters. The two properties of a Turing machine - unlimited time and unlimited space - are essential.

Let T, the **tape alphabet** be a finite set of characters. Let R be a finite set of **return values**. This is commonly just a singleton set, which would be analagous to `void` in Java, and simply tells us whether the prorgam returns (halts, terminates) or runs forever. It may also be a larger set, like a two-element set analagous to `boolean` for example. A Turing machine over T and R consists for the following data:

- A finite set Q of states
- An initial state $q_0 \in Q$
- A transition function δ from Q to the following set of behaviours:
  - $(T \to Q)$ (read from current position)
  - $(T \times Q)$ (write to current position)
  - Q (move left)
  - Q (move right)
  - Q (do nothing)
  - R (stop)

9 Church's Thesis

'What ever can be done by any computational device can be done by a Turing machine'

Church's thesis cannot be proven - it's entirely possible for someone to invent a new way of computation which extends the capabilities of programs as we know them. On the other hand, nobody has been able to come up with an alternative to Turing machines - the more time that passes to more plausible Church's thesis becomes.

10 Complexity and P

11 NP

12 Lambda-calculus

Alonzo Church developed a notation for arbitrary **functions** called λ-calculus. It is an extremely economical notation but at first sight somewhat cryptic, which stems from its origins in mathematical logic. Expressions in λ-calculus are written in strict prefix form. Further, function and argument are simply written next to each other without brackets around the argument.

12.1 Church-Rosser Theorem

'If a term M can be reduced (in several steps) to terms N and P, there there exists a term Q to which both N and P can be reduced (in several steps)

This is sometimes known as confluence. Regardless of the order in which we decide to reduce our lambda calculus expression, we will always be able to arrive at the same term. This makes perfect sense - when evaluating an expression with arithmetic terms, we cannot get two different answers dependent on the order of evaluation. It follows that we get this corollary:

'Every λ-term has at most one normal form'.

We can prove this. Assume there are two normal forms (for the case of contradiction) N and P to which are certain term M reduces. By the theorem of Church and Rosser there is a term Q to which both N and P can be reduced to. However, as we have assumed N and P are in their normal forms they don't allow any further reductions. The only possible interpretation for this is that N = P = Q.

12.2 Typed Lambda-calculus

Currently there is nothing in the grammar of lambda calculus that restricts us from forming awful terms. It would be completely possible to form sin log at the moment - the sine function applied to the logarithm function. This is obviously impossible and any programming language would reject this as incorrectly formed. We are currently missing the notion of types, which represent what kind of arguements the function will accept and what it expects to return.

We can form the language for expressing these types very easily. We begin with base types such as `int` and `real`. On top of this we then form function types. This gives us the following grammar which is called the **system of simple types**:

$$\tau ::= c \mid \tau \to \tau$$

In this grammar, the value c is used as a placeholder for all of our base types we wish to include. Now we can form function types, we would give the type of the sine function as `real` $\to$ `real`, which makes it obvious that it cannot accept the logarithm function as an argument. Using the type system that we have defined, we can create restrictions on what kind of terms are valid (or **well-typed**). We do this by an inductive definition:

- **Base Case**: For every type σ and every variable x, the term $x : \sigma$ is well typed and has type σ.
- **Function formation**: For every term M of type τ, every variable x, and every type σ, the term $\lambda x : \sigma.M$ is well-typed and has type $\sigma \to \tau$
- **Application**: If M is well-typed of type $\sigma \to \tau$ and N is well-typed of type σ then MN is well-typed and has type τ

List of Figures

Index