What's an Algorithm?

Adam Wyner
CS3518, Spring 2017
University of Aberdeen

Formal models of computation

- We briefly examined the lambda calculus, a formal model underlying Haskell.
- We shall explore another model of computation (called the imperative model), which underlies languages such as FORTRAN, PASCAL, C, JAVA etc.
- The name of this model: Turing Machines (TMs).

- Why study *formal models* (instead of only specific implementations)?

Formal models of computation

- "Formal models of computation" makes precise what it means for a problem to be solvable by means of computation:
  - Which problems are "computable"?
- The question is best addressed with a small programming language.
- The answer does not depend on the type of programming language (functional / logical / imperative); they are all equivalent.
- Church's Thesis: All languages formalise the notion of an algorithm.

The definition of an algorthim

- Informally, an algorithm is:
  - A sequence of instructions to carry out some task;
  - A procedure or a “recipe”
- Algorithms have had a long history in maths:
  - Find prime numbers
  - Find greatest common divisors (Euclid, ca. 300 b.c.)
- “Algorithm” was not defined until recently (1900’s)
  - Before that, people had an intuitive idea of algorithm
  - This intuitive notion was insufficient to gain a better understanding of algorithms

Algorithms in mathematics

- Next: how the precise notion of algorithm was crucial to an important mathematical problem…

Hilbert's problems

- David Hilbert gave a lecture
  - 1900
  - International Congress of Mathematicians, Paris
  - Proposal of 23 mathematical problems
  - Challenges for the coming century
  - 10^{th} problem concerned algorithms
- Before we talk about Hilbert's 10^{th} problem, let's briefly discuss polynomials…

Polynomials

- A polynomial is a sum of terms, where each term is a product of variables (which may be exponentiated by a natural number) and a constant called a coefficient.

- For example:

$$6x^3yz^2 + 3xy^2 - x^3 - 10$$

Terms: $6x^3yz^2$, $3xy^2$, x^3, 10
Variables: x, y, z
Coefficient: 10

Polynomials

- A root of a polynomial is an assignment of values to variables so the polynomial equals 0

$$6x^3yz^2 + 3xy^2 - x^3 - 10$$

- For example, if $x = 5$, $y = 3$ and $z = 0$, then

$$(6\times5^3\times3\times0^2) + (3\times5\times3^2) - 5^3 - 10 = 0$$

- This is an integral root; all variables assigned integers.
- Some polynomials have an integral root, some do not
  - $x^2 - 4$ has integral root
  - $x^2 - 3$ has no integral root

Hilbert's 10th problem

- Devise an algorithm that tests if a polynomial has an integral root
  - Hilbert wrote "a process according to which it can be determined by a finite number of operations"
  - Hilbert assumed such an algorithm existed; we only need to find it!
- We now know no algorithm exists for this task. It is algorithmically unsolvable; it is not computable.
- Impossible to conclude this with only an intuitive notion of algorithm!
  - Proving that an algorithm does not exist requires a clear definition of algorithm.
  - Hilbert's 10th Problem had to wait for this definition…

Church-Turing Thesis

- The definition of algorithm came in 1936
  - Papers by Alonzo Church and Alan Turing
- Church proposed the λ-Calculus
- Turing proposed Turing Machines
- They provide automata that are more powerful than finite state automata (details to follow!)
- These two definitions were shown to be equivalent, giving rise to the Church-Turing thesis.
- Thesis enabled a solution of Hilbert's 10^{th} problem.
  - Matijasevic (1970) showed no such algorithm exists

Recursively enumerable

- A language L is called recursively enumerable if and only if there exists an algorithm that, in response to the "Is x an element of L?":
  - says "yes" for precisely all the elements x of L.
  - does not say "yes" to anything else.
- If x is not an element of the language, then there are two possibilities:
  1. The algorithm says no to x;
  2. The algorithm does not find an answer.
- So, the algorithm might go on and on for elements presented but not in the language.

Decidable

- We prefer algorithms that always find an answer
  - They are called deciders.
  - A language for which there exists a decider is called decidable. (We also say: the problem is decidable).
  - A decider decides the language consisting of all the strings to which it says "yes".
  - A decider for a language L answers every question of the form "Is this string a member of L?" in finite time.
  - Algorithms that are not deciders keep you guessing on some strings.
- We shall soon see how these ideas pan out in connection with Turing Machines.
- First, Hilbert's 10^{th} problem.

Hilbert's 10th problem as a formal language

- Consider the set

  $D = \{p \mid p \text{ is a polynomial with an integral root}\}$
- Is *D* decidable?
  - Is there an algorithm that decides it?
- The answer is "no"!
- However, *D* is recursively enumerable
- To prove this, all we need to do is to supply an algorithm that "does the deed":

  - The algorithm will say "yes" if we input a polynomial that belongs to *D*, but it may loop if the polynomial does not belong to *D.*

A simpler version of the 10th problem

Polynomials with only one variable:

$$4x^3 - 2x^2 + x - 7$$

D_1= {p | p is a polynomial over x with an integral root}

An algorithm for D_1:

M_1 =

- The input is a polynomial *p* over *x*.
- Evaluate *p* with *x* set successively to the values

  0, 1, -1, 2, -2, 3, -3, …
- If at any point the polynomial evaluates to 0, say “yes”!

A simpler version (continued)

- If p has an integral root, M_1 will eventually find the root and accept p
- If p does not have an integral root, M_1 will run forever!
- It's like the problem of determining (e.g. in Haskell) whether a given infinite list contains 0.

A simpler version (continued)

- For the multivariable case, a similar algorithm M_2
  - M_2 goes through all possible integer values for each variable of the polynomial….
  - Suppose we have just two variables x and y:
    - fix x to 1, then vary y over integers till a solution...
    - fix x to 2, then vary y over integers till a solution...
    - .... or not!
- Both M_1 and M_2 are algorithms, but not deciders.
- Matijasevic showed the 10^{th} problem has no decider.
- This proof is omitted here. (But we shall prove another non-computability result later.)

Aside: converting M_1 into a decider

- The case with only 1 variable is decidable
- We can convert M_1 into a decider:
  - One can prove that the root x must lie between the values
    $$\pm\ k\ (c_{max}/c_1)$$
    where k is the number of terms in the polynomial
    c_{max} is the coefficient with largest absolute value
    c_1 is the coefficient of the highest order term
  - We only have to try a finite number of values of x.
  - If a root is not found within bounds, the machine rejects.

- Matijasevic: impossible to calculate bounds for multivariable polynomials.

Turing machines and algorithms

- Our focus is on algorithms.
- But it will help to have a more concrete idea of what an algorithm is.
- FSAs can be seen as algorithms, because they tell us which strings are elements of a language.
- However, FSAs do not allow us to do all the things that algorithms can do.
- Turing Machines (TMs) are much stronger automata.
- TMs are seen as faithful models for algorithms.
  - This is Church's thesis.

Philosophical questions about algorithms

- Is an algorithm that's not a decider (because on some inputs it does not terminate) really an algorithm?
  - Standard answer: yes. See the algorithm M1 above.
- Is an algorithm that never terminates an algorithm?
  - Standard answer: yes. Consier Haskell's function for the sequence of all Prime numbers. Think of lambda terms that do not have a normal form.
- Can an algorithm be stochastic?
  - Standard answer: yes. Think of nondeterministic FSAs. TMs can also be deterministic, and these TMs are generally thought of as algorithms.

Uncomputable problems

- We saw: if the answer to a yes/no problem can always be computed in finite time then the problem is called decidable.
- A more general term is “computable”: if the answer to a problem can always be computed in finite time then the problem is called computable.

Uncomputable problems

- Computability is about what can be computed at all (in finite time).
- Before going into computability, let's look at what can be computed in a given programming language.
- First let's prove in the abstract that some problems cannot be programmed in Haskell.
  - The same method could be employed to show that some problems cannot be programmed in JAVA, etc.
  - We will employ the same method to show that some problems cannot be programmed in a TM.

Theorem: some problems are not computable in Haskell

Outline of proof:

1. There are uncountably many problems.
2. There are countably many Haskell (JAVA, etc) programs.
3. Conclusion: Some problems cannot be programmed in Haskell (JAVA, etc.)

Let's look at this argument in more detail.

First: there are countably many Haskell programs

- Each Haskell program consists of a finite number of characters (the characters in the functions).
- Each character is chosen from a finite set
  (the set {a,b,…,z, A,B,..,Z, 0,1,2,..,9,' ,' ,(,),[,],:,.,|,^,&&,*,+,-,space})
- So the set of Haskell programs can be enumerated
  - Enumerate all programs of length 1 character (if these exist),
  - Enumerate all programs of length 2 characters (if these exist),
  - … etc. *Much like enumerating all rational numbers!*
- Discard all strings that do not make a well-formed Haskell program.
- The result is infinite but countable (like N) ("countably infinite").
- Note the transformation to a language problem.

Generalise to other languages

- Our enumeration may contain functions that do not terminate (e.g., f may defined as f), so if you like, you can discard these too. This will make the resulting set even smaller, so it does not affect our proof.
- The same argument shows that there are only countably many possible JAVA programs, web pages, novels, ...
- This is because each of these has a finite alphabet and a finite (though unlimited!) length.

Second: there are uncountably many problems/ languages

- This is most easily shown by focussing on a particular class of problems. For example,
  - Consider all functions f :: Int -> Int
    Each defines a language of 3-tuples.
    For example (f_1,x,y) iff $f_1(x)=y$
  - The number of functions of this type is uncountable.
    Prove this using Cantor's diagonal argument.
  - Recall lecture on Infinite Sets and enumerations.

Second: there are uncountably many problems/ languages

- Try to enumerate all functions f_i; each $f_i(j)$ is defined for every integer j.
- Define a new function g as $g(j) = f_i(j + 1)$, where i and j range incrementally over the integers.
- g cannot be an element of the enumeration since it is different from any f_i.

Uncountably many f :: Int -> Int

```
   1  2  3  4  5  6  7  8  ...
f1 .
f2    .
f3       .
f4          .
...
```

g differs from f1 on the argument 1

g differs from f2 on the argument 2

... *etc.*

So: g is not in the enumeration!

Where we are now

- You've seen a very abstract proof that some problems cannot be programmed in Haskell.
- More specifically, you've seen that some functions
  f :: Int -> Int cannot be programmed in Haskell
- But we haven't shown you a concrete example (e.g., a concrete problem for which there is no Haskell program).
- Is this a limitation of Haskell (JAVA,..)? No: We'll
  - Introduce TMs;
  - Argue that TMs can solve any computable problem;
  - Show that TMs can't program all problems.

Reading

- Introduction to the Theory of Computation. Michael Sipser. PWS Publishing Co., USA, 1997. (A 2nd Edition has recently been published). Chapter 3.
- Algorithmics: The Spirit of Computing. 3rd Edition. David Harel with Yishai Feldman. Addison-Wesley, USA, 2004. Chapter 9.