Slide 1

Models of Computation

Philippa Gardner

http://www.doc.ic.ac.uk/˜ pg/

We meet in room 311 on Wednesdays at 9–11 and in room 308 on Thursdays at 12-1. The tutorial will usually be on Wednesday at 10-11. I might move it around depending on what has been covered in the lectures. Please note that there are no lectures on 19th October due to the Commemoration Day.

Tutorial Helpers

Andrea Cerone, Petar Maksimović, Mark Wheelhouse, John Wickerson and Shale Xiong.

Introduction

This course provides a whirlwind tour of some of the key concepts in theoretical computer science. In particular, we will study formal descriptions (models) of computational behaviour. First, let's start with the notion of *algorithm*.

Slide 2

What is an algorithm?

A definition of algorithm from Chambers on-line dictionary:

> **algorithm** noun 1. any procedure involving a series of steps that is used to find the solution to a specific problem, e.g. to solve a mathematical equation. 2. *computing* the sequence of operations, often represented visually by means of a flow chart, that are to be performed by and form the basis of a computer program.
> **algorithmic** adjective.
> **Etymology**: 1930s in sense 1: from earlier *algorism* arithmetic, from Latin *algorismus*, named after Al-Khwarizmi, a 9th century Arab mathematician.

We can do better than this definition!

Examples of Algorithms One of the earliest algorithms was devised between 400 and 300 B.C. by Euclid: it finds the highest common factor of two numbers and is still used. The sieve of Eratosthenes is another old algorithm; look it up on Wikipedia, it's fun. Al-Khwarizmi is credited with devising the well-known rules for addition, subtraction, multiplication and division of ordinary decimal numbers.

Later examples of machines controlled by algorithms include weaving looms (1801, the work of J.M.Jacquard, 1752–1834), the player piano or pianoroll (the pianola, 1897—arguable as there is an analogue aspect), and the 1890 census tabulating machine of Herman Hollerith[1]. All these machines used holes punched in cards. In the 19th century, Babbage planned a *multi-purpose* calculating machine, the *analytical engine*, also controlled by punched cards.

[1] Hollerith is immortalised as the 'H' in the 'format' statement in the early programming language Fortran (e.g. FORMAT 4Habcd specified a four-character string 'abcd'). He also founded a company that turned into IBM.

Slide 3

Examples of Algorithms

Euclid's algorithm (approx 300 B.C.)

The Sieve of Eratosthenes (approx 200 B.C.)

The well-known rules for addition, subtraction, multiplication and division of numbers (Al-Khwarizmi, 9th C)

Simple machines using punched cards: weaving looms, pianola?, census tabulating machine (mainly 19th C)

The proposed analytical machine, a multi-purpose calculating machine (Babbage, late 19th C)

Unsolvable Problems The great mathematician David Hilbert posed Hilbert's *Entscheidungsproblem* (slide 4) in the 1928 International Congress of Mathematicians. Such an algorithm would be invaluable. For example, by running it on

$$\forall k > 1. \exists p, q. \, (2k = p + q \wedge \mathtt{prime}(p) \wedge \mathtt{prime}(q)),$$

where $\mathtt{prime}(p)$ is a suitable arithmetic statement that p is a prime, we could solve the famous, open problem, *Goldbach's conjecture*, that every strictly positive even number is the sum of two primes. Hilbert thought that such an algorithm could be found. In 1935/36, he was proved wrong by the work of Turing and Church.

Slide 4

Hilbert's *Entscheidungsproblem*

> Is there an algorithm which, when fed any statement in the formal language of first-order logic, determines in a finite number of steps whether or not the statement is provable, using the usual rules of first-order logic?

The problem was originally posed in a more ambiguous form, with a more powerful formal system in place of first-order logic.

Slide 5

Decision Problems

Entscheidungsproblem means 'decision problem'.

Given

- a set S whose elements are finite data structures of some kind: e.g., formulae of first-order logic
- a property P of elements of S: e.g., the property of a formula that it has a proof,

the associated decision procedure is

> find an algorithm which terminates with result 0 or 1 when given an element $s \in S$, and yields result 1 when given s if and only if s has property P.

Slide 6

Algorithms, informally

People tried to find such an algorithm to solve Hilbert's Entscheidungsproblem, without success.

A natural question was then to ask whether it was possible to prove that such an algorithm did not exist. To ask this question properly, it was necessary to provide a formal definition of algorithm.

Common features of the examples of algorithms:

- finite description of the procedure in terms of elementary operations;
- deterministic, next step is uniquely determined if there is one;
- procedure may not terminate on some input data, but we can recognise when it does terminate and what the result will be.

Why did no-one give a precise definition of algorithm in the preceding two thousand years? Perhaps because most questions on algorithms are of the form 'find an algorithm to solve this problem I have'. This can be done without a formal definition of algorithm, because we know an algorithm when we see one. Just as an elephant is easy to recognise but hard to define, we can write a program to sort a list without knowing *exactly* what an algorithm is. It is enough to invent something that intuitively is an algorithm, and that solves the problem in question. We do it all the time.

But suppose we had a problem (like Hilbert's Entscheidungsproblem) for which many attempts to find an algorithmic solution had failed. Then we might suspect that the task is impossible, so we would like to *prove* that no algorithm solves the problem. To have any hope of doing this, it is essential to define precisely what an algorithm is. Similarly, to answer questions concerning *all* algorithms, we need to know exactly what an algorithm is. Otherwise, how could we proceed at all?

Slide 7

Algorithms, informally

In 1935/36, Turing in Cambridge and Church in Princeton independently gave negative solutions to Hilbert's Entscheidungsproblem.

- First they gave a precise, mathematical definition of algorithm: Turing invented **Turing machines** and Church invented the **lambda calculus**.
- Then they regarded **algorithms as data** on which algorithms can act, and reduced the problem to ...

Slide 8

Algorithms, informally

In 1935/36, Turing in Cambridge and Church in Princeton independently gave negative solutions to Hilbert's Entscheidungsproblem.

- First they gave a formal definition of algorithm: Turing invented **Turing machines** and Church **lambda calculus**.
- Then they regarded **algorithms as data** on which algorithms can act, and reduced the problem to **the Halting Problem**.

Slide 9

Algorithm formalised

Any formal definition of an algorithm should be

- **precise** and unambiguous, with no implicit assumptions, so we know what we are talking about; for maximum precision, it should be phrased in the language of mathematics;
- **simple** and without extraneous details, so we can reason easily with it;
- **general**, so that all algorithms are covered.

Turing introduced the **Universal Turing machine**, a Turing machine that can simulate an arbitrary Turing machine on arbitrary input. This work led to the **Church-Turing Thesis** that, in basic terms, everything computable is computable by a Turing machine.

Pay-offs from the formalism of algorithm include the modern programmable computer itself! [1] Other payoffs include the identification of unsolvable questions such as Hilbert's Entscheidungsproblem, and work on the design of beautiful and efficient algorithms for solving everyday problems (including heuristics for the computationally 'hard' problems), analysing issues such as time, memory, I/O etc. You have a course on algorithms in the spring term.
Turing's formalism was given using a primitive computer called (nowadays) the *Turing machine*. The Turing machine first appeared in his paper in 1936, some ten years before 'real' computers were invented. Turing's definition of algorithm was that an algorithm is what a Turing machine implements. We will define the Turing machine in this course. We will see that it is *precise* and *simple* [2], just as a formalism should be. However, to claim that it is *fully*

[1] This is an arguable historical point: see Hodge's book for a good account of the historical background. Turing himself clearly played his part in the development of programmable computers. During the Second World War, Turing worked for the Government Code and Cypher School at Bletchley Park, Britain's code-breaking centre. For a time he was head of Hut 8, the section responsible for German naval cryptanalysis. He devised a number of techniques for breaking German ciphers, including the method of the Bombe, an electromechanical machine that could find settings for the Enigma machine. After the war he worked at the National Physical Laboratory, where he created one of the first designs for a stored-program computer, the ACE.

[2] Perhaps this is open to debate!

general–covering all known and indeed all conceivable algorithms–is a pretty strong claim, especially when we see how primitive the Turing machine is. Turing gave substantial evidence for this general claim in his paper, evidence that has strengthened over the years, and the typical view nowadays is that the Turing machine is fully general. For historical reasons, this view is known as *Church's thesis*, or sometimes (better) as the *Church-Turing thesis*.

Slide 10

The Halting Problem

The Halting Problem is the decision problem with

- the set S of all pairs (A, D), where A is an algorithm and D is some input datum on which the algorithm is designed to operate;
- the property $A(D)\downarrow$ holds for $(A, D) \in S$ if algorithm A when applied to D eventually produces a result: that is, eventually halts.

Turing and Church's work shows that the Halting Problem is unsolvable (undecidable): that is, there is no algorithm H such that, for all $(A, D) \in S$,

$$\begin{aligned} H(A, D) &= 1 \quad A(D)\downarrow \\ &= 0 \quad \text{otherwise} \end{aligned}$$

Slide 11

From HP to *Entscheidungsproblem*

The final step in the Turing/Church proof of the unsolvability of the *Entscheidungsproblem* constructs an algorithm encoding instances (A, D) of the Halting Problem as statements $\Phi_{A,D}$ with the property

$$\Phi_{A,D} \text{ is provable } \leftrightarrow A(D) \downarrow$$

Thus, any algorithm which decides the provability of such statements could be used to decide the Halting Problem. Hence, no such algorithm exists.

With hindsight, a positive solution to *Entscheidungsproblem* would have been too good to have been true. However, the algorithmic unsolvability of some decision problems is much more surprising. A famous example of this is Hilbert's 10th Problem on Diophantine equations. A *Diophantine equation* has the form

$$p(x_1, ..., x_n) = q(x_1, ..., x_n)$$

where p and q are polynomials in unknowns $x_1, ..., x_n$ with coefficients from the set of natural numbers $\{0, 1, 2, 3, ...\}$. These equations are named after Diophantus of Alexandria (approx 250 A.D.). For example, 'find three whole numbers x_1, x_2 and x_3, such that the product of any two added to the third is a square' [Diophantus' *Arithmetica*, Book III, Problem 7].

Slide 12

Hilbert's 10th Problem

Give an algorithm which, when given a Diophantine equation, determines in a finite number of operations whether or not there are natural numbers satisfying the equation.

This was listed by Hilbert as a challenge problem in 1900. It was only resolved in the 1970s, by Matijasevic, Robinson, Davis and Putnam, who showed it was unsolvable by reduction to the Halting Problem.

The original proof used Turing machines. Later, a simpler proof (Jones and Matijasevic, Journal of Symbollic Logic, 49(1984)) used Minsky and Lambek's register machines.

Register machines are more like the computers we use every day, compared with Turing's tape machines.

Slide 13

Algorithms as Special Functions

Turing and Church's equivalent definitions of algorithm capture the notion of computable function: an algorithm expects some input datum D, does some calculation and, if it terminates, returns a unique result.

Question Is it possible to give a mathematical description of a computable function as a special function between special sets?

Slide 14

Algorithms as Functions

Question Is it possible to give a mathematical description of a computable function as a special function between special sets?

Answer Yes, but it took a long time to work it out.

In the 1960s, Strachey and Scott in Oxford introduced **denotational semantics**, which describes the **meaning (denotation)** of an algorithm as a function that maps input data to an output result.

In particular, Scott solved the difficult part, giving meaning to recursively-defined algorithms as continuous functions between domains (special sets).

Denotational semantics originated in the work of Christopher Strachey and Dana Scott at Oxford in the 1960s. Denotational semantics provides the denotation (meaning) of an algorithm, for example as a function that maps inputs to outputs. To give denotations to recursively-defined algorithms, Scott proposed working with continuous functions between domains: more specifically, complete partial orders. We shall see that an important property of denotational semantics is that the semantics is compositional: the denotation of a program is built out of the denotations of its program fragments.
In fact, there are three key forms of program semantics: denotational semantics, operational semantics and axiomatic semantics. To motivate these semantics, let's look at two simple Haskell programs.

Slide 15

Two Haskell Programs

What is the behaviour (meaning) of these two Haskell programs?

```
power x n
 | n == 0 = 1
 | otherwise = x * power x ( n - 1 )

power' x n
 | n == 0 = 1
 | even n = k^2
 | odd n  = x*k^2
 where
   k = power' x (n `div` 2)
```

Slide 16

Two Haskell Programs

The Haskell functions `power` and `power'` can be viewed as equivalent: given any two inputs x and n, they compute the **same** result x^n.

Operationally, they are different.

Slide 17

Two Haskell Programs

Example

```
power 3 5
 -> 3 * (power 3 4)
 -> 3 * (3 * (power 3 3))
 -> 3 * (3 * (3 * (power 3 2)))
 ->  3 * (3 * (3 * (3 * (power 3 1))))
 ->  3 * (3 * (3 * (3 * (3 * (power 3 0)))))
  ->  3 * (3 * (3 * (3 * (3 * 1))))
```

(`power x n`) takes `O(n)` steps.

Slide 18

Two Haskell Programs

Example

```
power' 3 5
 -> 3 * (power'3 2)^2
 -> 3 * ((power' 3 1)^2)^2
 -> 3 * ((3 * (power' 3 0)^2)^2)^2
 -> 3 * ((3 * 1^2)^2)^2
```

`(power' x n)` takes `O(log n)` steps.

Slide 19

Program semantics

Denotational Semantics: a program's meaning is described *compositionally* using mathematical objects (called *denotations*): the denotation of a program phrase should be built out of denotations of its subphrases.

Operational Semantics: a program's meaning is given in terms of the steps of computation the program makes when it runs.

> It is all very well to aim for a more abstract and a cleaner approach to semantics, but if the plan is to be any good, the operational aspects cannot be completely ignored (Scott70).

There is another key style of program semantics, called **axiomatic semantics**, which we do not study in this course.

Operational semantics provides a precise, mathematical description of the computational behaviour of programs. Many programming languages are currently described using informal *natural language*: for example, the English standards documents for C, Java, JavaScript, ... These documents are reasonably accessible (though often written in 'standardese'), but there are some major problems. It is hard, if not impossible, to write precise definitions in informal prose. The standards often end up being ambiguous or incomplete, or just too large and difficult to understand. This leads to differing implementations and flaky systems, as the language implementers and users do not have a common understanding of what the language is. More fundamentally, natural language standards obscure the real structure of language. It is all too easy to add a feature and a quick paragraph of text without thinking about how it interacts with the rest of the language.

For a long time, we have been able to provide operational semantics for featherweight fragments of programming languages to explore how specific program features combine. More recently, we have been able to provide the operational semantics of whole real-world program languages. The functional programming language *Standard ML* was the first language to have a complete formal operational semantics. It was written in 1990 and revised in 1997. It was mechanised in the proof assistant Twelf in 2006. Other functional languages (Haskell, CAML and F♯) do not have complete formal definitions. However, all the inventors of these languages have looked at the operational semantics for featherweight fragments. There have been many partial definitions of the C language using operational semantics. Of particular note is the CompCert project of Leroy, which formally defined CLight (a large fragment) in the Coq theorem prover and used it to verify a realistic compiler. More recently, Ellison and Rosu have given a mechanised definition of the full C language. Several Java fragments (some featherweight, some quite large) have been studied, initiated by work of Drossopoulou and Eisenbach, Imperial. The .NET Common Language Runtime (CLR) has a formal semantics. XQuery has a formal standardized W3C specification, although recent extensions have not been formally specified. Maffeis, Imperial, and Mitchell and Tally, Stanford, have given a complete semantics of Javascript (ES3), which has been used to identify secure fragments For example, they showed that the Facebook fragment was insecure and suggested the fix. Recently, many of us at Imperial and INRIA, France, have given a mechanised semantics of JavaScript (ES5) in Coq, with a reference interpreter that has been proved correct with respect to the semantics and tested using the Test 262 test suite.

Slide 20

Informal Programming-Language Description

An extract from the Algol 60 report:

> Finally the procedure body, modified as above, is inserted in place of a procedure statement and executed. If a procedure is called from a place outside the scope of any non-local quantity of the procedure body, the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

Algol 68 is described using operational semantics.

Algol 60 is a relatively nice language. Imagine how complex the English language specification for JavaScript is!

Slide 21

Examples of Operational Semantics

Standard ML: formal definition Milner, Harper, MacQueen, Tofte, 1990 and 1997, mechanised definition Lee, Crary, Harper, 2006.

C: many partial definitions, some mechanised; complete mechanised definition POPL 2012, Ellison, Rosu

Featherweight fragments of Java: many partial (large) definitions, first studied by Drossopoulou, Eisenbach, 1997, mechanised definition Syme, 1998.

XQuery: formal W3C standard.

Javascript: complete formal definition ES3, used to identify secure fragments, Maffeis (Imperial) , and Mitchell and Tally (Stanford), mechanised definition ES5 many at Imperial and INRIA, POPL 2014

Slide 22

This Course

- Operational semantics of a small while language
- Denotational semantics of a small while language
- Register machines, the universal register machine, the Halting Problem
- Turing machines and Turing computable functions, primitive and partial recursive functions, lambda calculus, equivalence results.

References

H.R. Nielson and F. Nielson (1999). *Semantics with Applications: A Formal Introduction*, originally published in 1992 by John Wiley and Sons, revised edition on web (see my web page).

G. Winskel (1993). *The Formal Semantics of Programming Languages*, MIT Press. This is an excellent introduction to both the operational and denotational semantics of programming languages.

M. Hennessy (1990). *The Semantics of Programming Languages*, Wiley, revised edition on web (www.cogs.susx.ac.uk/users/matthewh/semnotes.ps.gz). The book is subtitled 'An Elementary Introduction using Structural Operational Semantics', and provides a leisurely introduction to some of the topics in this course.

J.E. Hopcroft, R. Motwani and J.D. Ullman (2001). *Introduction to Automata Theory, Languages and Computation*, 2nd edition, Addison-Wesley.

J.R. Hindley and J.P. Seldin (2008). *Lambda Calculus and Combinators, an Introduction*, 2nd edition, Cambridge University Press.

N,J. Cutland (1980). *Computability. An Introduction to Recursive Function Theory*, Cambridge University Press.

M.D. Davis, R. Sigal and E.J. Wyuker. (1994). *Computability, Complexity and Languages*, 2nd edition, Academic Press.

T.A. Sudkamp (1995). *Languages and Machines*, 2nd edition, Addison-Wesley.

Notes The material in these notes has been drawn from several different sources, including the books mentioned above, previous courses on *Models of Computation* at Imperial by Steffen van Bakel, Dirk Pattinson and Chris Hankin, a previous course on *Computability, Algorithms and Complexity* by Ian Hodkinson, and courses at some other universities. I would particularly like to mention courses by Andy Pitts on *Computation Theory* at Cambridge, by Peter Sewell on *Semantics of Programming Languages* at Cambridge, and by Matthew Hennessey and Guy McCusker also on *Semantics of Programming Languages* . Any errors are of course my own.

Slide 23

Assessment

- Assessed course work, published on Wednesday 9th November and submitted on Friday 18th November by 2 pm.
- Assessed course work, distributed on Wednesday 7th December and submitted on Friday 16th December by 2 pm.
- An exam paper in the summer term, shared with the compilers course: four compulsory questions; two on *Models of Computation*.

In addition, unassessed exercises will be given throughout the course.

The proposed dates of the assessed course work might vary, so that you do not have too many deadlines in one week.

Student reps, please give feedback on the dates.