Course Aims

▶ understanding of computability, complexity and intractability;
▶ knowledge of lambda calculus, types, and type safety.

Course Outcomes

By the end of the course you should be able to

▶ Explain decidability, undecidability and the halting problem.
▶ Demonstrate the use of reductions for undecidability proofs.
▶ Explain the notions of P, NP, NP-complete.
▶ Use reductions to show problems to be NP-hard.
▶ Write short programs in lambda-calculus.
▶ Explain and demonstrate type-inference for simple programs.

Course Outline

This outline may change slightly, but is basically sound.

▶ Introduction. The meaning of computation.
▶ Register machines and their programming.
▶ Universal machines and the halting problem.
▶ Decision problems and reductions.
▶ Undecidability and semi-decidability.
▶ Complexity of algorithms and problems.
▶ The class P
▶ Non-determinism and NP
▶ NP-completeness
▶ Beyond NP.
▶ Lambda-calculus.
▶ Recursion.
▶ Types.
▶ Polymorphism.
▶ Type inference.
Assessment

The course is assessed by a written examination (70%) and two coursework exercises (15% each).

Coursework deadlines: 16:00 on 13 February and 20 March (end of weeks 5 and 9 – ILW isn’t a numbered week!).

The courseworks will, roughly, contain one question for each week of the course. Please do the questions in/following the week to which they apply, rather than waiting till the deadline to do them all!

Lectures

I see the purpose of lectures as providing the key concepts on to which you can hook the rest of the material.

Slides are few in number, and text-dense: I will spend a lot of time talking about things and using the board, with the slides providing the reminder of what we’re doing.

Interaction in lectures is encouraged!

From time to time, we’ll do relatively detailed proofs, but mostly this will be left to you in exercises and independent study.

Textbooks

It will be useful, but not absolutely necessary, to have access to:

- Benjamin C. Pierce *Types and Programming Languages*, MIT Press

There is also much information on the Web, and in particular Wikipedia articles are generally fairly good in this area.

Generally I will refer to textbooks for the detail of material I discuss on slides.

What is computation? What are computers?

Some computing devices:

The abacus – some millennia BP.

[Association pour le musée international du calcul de l’informatique et de l’automatique de Valbonne Sophia Antipolis (AMISA)]
First mechanical digital calculator – 1642 Pascal


What do computers manipulate?
Symbols? Numbers? Bits? Does it matter?
What about real numbers? Physical quantities? Proofs? Emotions?
Do we buy that numbers are enough? If we buy that, are bits enough?
How much memory do we need?

Analytical Engine (never built) anticipated many modern aspects of computers. See http://www.fourmilab.ch/babbage/.

ENIAC – 1945, Eckert & Mauchley
What can we compute?
If we can cast a problem in terms that our computers manipulate, can we solve it? Always? Sometimes? With how much time? With how much memory?

Register Machines
At the end of Inf2A, you saw Turing machines and undecidability. We’ll reprise this using a different model a bit closer to (y)our ideas of what a computer is.
The simplest Register Machine has:
▶ a **fixed** number of registers $R_0, \ldots, R_{m-1}$, which each hold a **natural number**;
▶ a **fixed** program that is a sequence $P = I_0, I_1, \ldots, I_{n-1}$ of instructions.
▶ Each instruction is one of:
  - **INC**$(i)$: add 1 to $R_i$, or
  - **DECJZ**$(i, j)$: if $R_i = 0$ then go to $I_j$, else subtract 1 from $R_i$
▶ The machine executes instructions in order, except when **DECJZ** causes a jump.

CLAIM: Register Machines can compute anything any other computer can.

Programming RMs
RM are very simple, so programs become very large.
First of all, we’ll define ‘macros’ to make programs more understandable. These are **not** functions or subroutines as in C or Java: they are like #define in C, **human** abbreviations for longer bits of code.
We’ll write them in English, e.g. ‘add $R_i$ to $R_j$ clearing $R_i$.
When we define a macro, we’ll number instructions from zero. When the macro is used, the numbers must be updated appropriately (‘relocated’).
We’ll also use symbolic labels for instructions, and use them in jumps.
We can then build macros into the hardware by adding new instructions that may have access to special registers: we’ll use negative indices for registers that are not to be used by normal programs.

**Convention:** macros must leave all special registers they use at zero – and hence the special registers can be assumed to be zero on entry.

Easy RM programs
▶ ‘goto $I_j$’ using $R_{-1}$ as temp
  0 **DECJZ** $(-1, j)$
▶ ‘clear $R_i$’
  0 **DECJZ** $(i, 2)$
  1 **GOTO** 0
▶ ‘copy $R_i$ to $R_j$’ using $R_{-2}$ as temp
  0 **CLEAR** $R_j$
  **LOOP1**: 2 **DECJZ** $(i, \text{loop2})$
  3 **INC** $(j)$
  4 **INC** $(-2)$
  5 **GOTO** **loop1**
  **loop2**: 7 **DECJZ** $(-2, \text{end})$
  8 **INC** $(i)$
  9 **GOTO** **loop2**
  **end**: 10

What is unrealistic about these machines?
RM programming exercises

- Addition/subtraction of registers
- Comparison of registers
- Multiplication of registers
- (integer) Division/remainder of registers

Towards a universal RM

Programs are data.
Can we build an RM that can interpret any other RM?
Need to encode RMs as numbers. What do we need for a machine $M$?

We need the contents $R$ of the registers $R_0, \ldots, R_{m-1}$, the program $P = l_0 \ldots l_{n-1}$ itself, and the ‘program counter’ $C$ giving the current instruction.

Let $\langle \ldots \rangle$ be the coding function given thus:

$\langle \text{INC}(i) \rangle = \langle 0, i \rangle$
$\langle \text{DEC}Z(i,j) \rangle = \langle 1, i, j \rangle$
$\langle P \rangle = \langle \langle l_0 \rangle, \ldots, \langle l_{n-1} \rangle \rangle$
$\langle R \rangle = \langle R_0, \ldots, R_{m-1} \rangle$
$\langle M \rangle = \langle \langle P \rangle, \langle R \rangle, \langle C \rangle \rangle$

Routine but very tedious Exercise: design an RM that takes an RM coding in $R_0$, simulates it, and leaves the final state (if any) in $R_0$.

How many registers?

So far, we have introduced registers as needed. Do we need machines with arbitrarily many registers?

A pairing function is an injective function $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$.

An easy mathematical example is $(x, y) \mapsto 2^x 3^y$.

Given a pairing function $f$, write $\langle x, y \rangle$ for $f(x, y)$. If $z = \langle x, y \rangle$, let $z_0 = x$ and $z_1 = y$.

Exercise: program pairing and unpairing functions in an RM.

Exercise: design (or look up) a surjective pairing function.

Can generalize to $\langle \ldots \rangle_n : \mathbb{N}^n \rightarrow \mathbb{N}$. (Exercise: give two different ways.)

And thence to $\langle \ldots \rangle : \mathbb{N}^* \rightarrow \mathbb{N}$

Thus we can compute with arbitrarily many numbers with a fixed number of registers.

With a couple of extra instructions (‘goto’ and ‘clear’), just two user registers are enough. (Think about what this means.)

and on to the halting problem

“the final state (if any)” …

Can we tell computationally whether (any) simulated machine halts, by some clever analysis of the program?

- Suppose $H$ is an RM $(P_H, R_0, \ldots)$ which takes a machine coding $\langle M \rangle$ in $R_0$, and terminates with 1 in $R_0$ if $M$ halts, and 0 in $R_0$ if $M$ runs forever.

- It’s then easy to construct $L = (P_L, R_0, \ldots)$, which takes a program (code) $\langle P \rangle$ and terminates with 1 if $H$ returns 0 on the machine $(P, \langle P \rangle)$, and itself goes into an infinite loop if $H$ returns 1 on $(P, \langle P \rangle)$.

- $(L$ takes a program, and runs the halting test on the program with itself as input (diagonalization), and loops iff it halts.)

- What happens if we run $L$ with input $\langle P_L \rangle$?

- If $L$ halts on $\langle P_L \rangle$, that means that $H$ says that $(P_L, \langle P_L \rangle)$ loops; and if $L$ loops on $\langle P_L \rangle$, that means that $H$ says that $(P_L, \langle P_L \rangle)$ halts. So either way, contradiction.
That (sketch) proof assumed very little – doesn’t actually need RMs, just a function of two inputs that can be understood as program and data. So why is it such a big deal?

Because we convinced ourselves (?) that RMs can compute anything that is computable in any reasonable sense. So we’ve shown that some things can’t be computed in any reasonable sense – which was not obvious.

But are Register Machines the right thing? Maybe other machines can do more?

Turing machines

(Reprise from Inf2A) Turing’s original paper (quite readable - read it!) used what we now call Turing machines.

A Turing machine comprises:

- a tape \( t = t_0 t_1 \ldots \) with an unlimited number of cells, each of which holds a symbol \( t_i \) from a finite alphabet \( A \);
- a head which can read and write one cell of the tape, and be moved one step left or right; let \( h \) be the position of the head;
- a finite set \( S \) of control states
- a next function \( n : S \times A \rightarrow (S \times A \times \{+1, -1\}) \cup \{\text{halt}\} \)

The execution of the machine is:

- Suppose the current state is \( s \), and the head position is \( h \), then let \((s', a', k) = n(s, t_h)\): then cell \( h \) is written with the symbol \( a' \), the head moves to position \( h + k \), and the next state is \( s' \);
- if \( n(s, t_h) = \text{halt} \), or \( h + k < 0 \), the machine halts.

Programming Turing machines

How to represent numbers? In binary, with 0 and 1? In unary, with just 1?

Typically use binary. Usually convenient to have a ‘blank’ symbol, and a few marker symbols such as $.

For further information and examples, see Sipser ch. 3.

Exercise: Design a TM (with any convenient alphabet) to interpret an RM program.

Exercise: Design an RM to interpret a TM specification.

(Don’t really do these – just think about how they can be done.)

A predecessor course

http://www.inf.ed.ac.uk/teaching/courses/ci/

provides a Java TM simulator and a bunch of TMs, including a universal TM!

TM variations

Bi-infinite tapes, multiple tapes, etc. may make programming easier, but aren’t necessary.

One semi-infinite tape and two symbols are needed.