3 Pairing and encoding functions

Our aim in this part of the course is to show that register machines can compute everything that can be computed, and to show that there are things that can’t be computed. To do this, we will need to express everything – including register machines themselves – as numbers, so that we can compute with everything. We will use the term *encoding function* to refer to a function that takes an object of some kind and represents it as a natural number. First, we’ll consider functions for encoding pairs, triples, etc. of numbers into a single number.

**Definition 5** A *pairing function* is an injective function \( \mathbb{N} \times \mathbb{N} \to \mathbb{N} \).

Given a pairing function \( f \), write \( \langle x, y \rangle_2 \) for \( f(x, y) \). If \( z = \langle x, y \rangle_2 \), let \( z_0 = x \) and \( z_1 = y \).

An easy example is \( (x, y) \mapsto 2x \cdot 3^y \). Note that this function is easy to invert: if I give you \( z = 2x \cdot 3^y \), you can just repeatedly divide by two and count to get \( x \), and then repeatedly by three and count to get \( y \). We will use this shortly.

We can generalize to functions that encode \( n \)-tuples of integers, which we write \( \langle \ldots \rangle_n : \mathbb{N}^n \to \mathbb{N} \). One way to do this is to generalize the above example:

\[
\langle x_1, \ldots, x_n \rangle_n = \prod_{1 \leq i \leq n} p_i^{x_i}
\]

where \( p_i \) is the \( i \)'th prime. We’ll use that below. Another way uses repeated pairing.

Even more generally, we can write functions \( \langle \ldots \rangle : \mathbb{N}^* \to \mathbb{N} \) to encode an arbitrary length sequence of integers into an integer, as we’ll see below.

4 Three into two does go

We’ll now show that we can easily encode a three-register machine program into a two-register machine program, given the three instructions \textsc{inc}, \textsc{decjz}, \textsc{goto}.

For this, we will use the ‘obvious’ tripling function:

\[
\langle x, y, z \rangle = 2^x \cdot 3^y \cdot 5^z
\]

The reason for this choice is that it is particularly easy to manipulate the three components by manipulating the code directly.

The target machine has three registers, which we call \( T_0, T_1, T_2 \); our interpreting machine has two, \( R_0, R_1 \). The principle of the encoding is that \( R_0 \) always contains \( \langle T_0, T_1, T_2 \rangle \), while \( R_1 \) is our work register.

Each instruction of the target machine is translated to a macro of the interpreting machine; we will now define these macros.

First, consider the instruction \textsc{inc} \( T_0 \). Incrementing \( T_0 \) corresponds to multiplying \( \langle T_0, T_1, T_2 \rangle \) by 2:
Equally easily, we can translate \( \text{inc} \ T_1 \) and \( \text{inc} \ T_2 \). The decrement instructions are a little trickier – to encode \( \text{decjz} \ T_0, \text{dest} \), we need to divide \( \langle T_0, T_1, T_2 \rangle \) by 2, but back out and jump if it turns out not to be a multiple of 2. Note, by the way, that \( R_0 \) can never contain zero on entry, as this is not a valid triple code.

Similarly, though rather more tediously, particularly with the recovery, we can write macros to decrement \( T_1 \) and \( T_2 \). The target \( \text{goto} \) is translated directly.

Modulo correctness of the above macros, we have now encoded a 3-register machine into a 2-register machine.

It is obvious that for any given value of \( n \), this encoding can be extended to encode
n-register machines into 2-register machines, and so in some sense we’re done: we’ve shown two registers are enough to encode any given machine, and so any RM-computable function can be simulated by a 2-register RM.

It is an interesting quirk that while a 2-register RM can simulate a general RM, and can therefore compute a suitable encoding of any computing function, it is not true that a 2-register RM can compute any computable function, in the sense of taking the input in $R_0$ and leaving the answer in $R_0$. Such simple functions as $n^2$, $2^n$ cannot be computed (in this sense) by a 2-register machine.

5 Simulating unbounded register numbers

However, it would be more satisfactory if we had a single encoding that would translate machines with arbitrary numbers of registers into 2-register machines. This is rather harder, because it means we have to treat the target register index ($i$ in $T_i$) as a variable quantity, instead of hard-wiring each different instance. Moreover, using the arbitrary extension of the prime-powers pairing function would involve us generating the $i$th prime on the fly every time we touched $T_i$. While this is possible, it’s pretty hideous (and certainly not obviously doable with two registers).

So in order to sketch the proof, I’ll assume some pairing function $\langle \cdot, \cdot \rangle_2 : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \setminus \{0\}$ (note that $2^x3^y$ is such a function), and then define the following coding of sequences:

\[
\langle \rangle = 0 \\
\langle x, s \rangle = \langle x, \langle s \rangle \rangle_2
\]

where $s$ is a sequence

Hence, extracting the $i$ element of a coded sequence is simply a matter of pulling out the first component of a pair, $i$ times. Assuming the unpairing function is reasonable, this can be done with a couple of auxiliary registers.

Similarly, updating the $i$th element is routine, though a little more tedious. Try writing the pseudo-code for it, and see how many auxiliary registers you need.

Given that, then the rest of the translation process is very simple: if interpreting machine $R_0$ holds the coded registers, then target machine instruction INC($i$) turns into LOAD $R_1$ with $i$

EXTRACT $T_i$ into $R_2$, saving $\langle T_0, \ldots, T_{i-1} \rangle$ in $R_3$ and leaving $\langle T_{i+1}, \ldots \rangle$ in $R_0$

INC ($R_2$)

PACK $T_0, \ldots, T_{i-1}, R_2, T_{i+1}, \ldots$ into $R_0$

The macros are a bit fiddly, especially the re-packing (if you are a Lisp programmer, you may remember basic exercises that involved a lot of list reversing), but not difficult.

Similarly for DECJZ and GOTO.

While I haven’t counted, I’d guess that ten registers are enough to do all that conveniently – possibly fewer.

But we wanted just two registers.

No problem! If we can produce an arbitrary-register-RM interpreter using ten registers, we can encode it into a two-register machine using the technique of the previous section.

Note, however, that if we do all this with the primes coding function, the machine has zilch chance of doing anything useful in the lifetime of the universe. What is the code of the sequence $(2, 2, 2, 2, 2)$?

For the purpose of this section, any coding function will do, so we can code more efficiently using the diagonal pairing function (or to be precise, $1 +$ the diagonal pairing function).
For the construction of section 4, I don’t know any tupling function that can be calculated with only one scratch register, and is not exponential in the inputs. I also don’t know that no such function exists.

I believe that the `goto` is necessary, in that without it we cannot write a general RM simulator using just two registers. So far, the proof escapes me.

6 Universal machines

In the previous sections, we showed that a particular three-register or \( n \)-register machine could be simulated by a particular two-register machine. Now we will show (without details) that we can build a single machine that can simulate any register machine. To do this, we need to encode RMs into numbers. Following a common convention in logic, we shall use \( \langle \cdot \rangle \) to denote a coding function into the integers. It is important to remember that \( \langle \cdot \rangle \) is just an integer, whatever the \( \cdot \) are. The coding function is as on the slides:

**Definition 6** Let \( M \) be a Register Machine, and let \( R \) be the contents of registers \( R_0, \ldots, R_{m-1} \), \( P = I_0 \ldots I_{n-1} \) be the program itself, and let \( C \) be the program counter, as in definition 2. Let \( \langle \cdot \rangle \) be a sequence coding function.

We define the RM coding function \( \langle \cdot \rangle \) by:

\[
\begin{align*}
\langle \text{INC}(i) \rangle &= \langle 0, i \rangle \\
\langle \text{DECJZ}(i, j) \rangle &= \langle 1, i, j \rangle \\
\langle P \rangle &= \langle \langle I_0 \rangle, \ldots, \langle I_{n-1} \rangle \rangle \\
\langle R \rangle &= \langle R_0, \ldots, R_{m-1} \rangle \\
\langle M \rangle &= \langle P, R, C \rangle
\end{align*}
\]

We now build a universal machine in a very similar way to the previous simulation of \( n \)-register machines by 2-register machines. The main difference is that instead of hard-wiring the translation of each program instruction, we extract the code of the instruction from the coded program, and translate it ‘in software’ by examining the code.

As remarked on the slides, this is not hard, but it is a substantial programming task. If you want to see a fairly detailed description of the construction, my colleague Philippa Gardner at Imperial College did a lecture about it. There are some small differences in notation, and her ‘gadgets’ are my ‘macros’, but it should be easy to follow. Her lecture is here:


If we do this work, we obtain

**Theorem 7** There is a Register Machine \( U \) which takes as input the code \( \langle M \rangle \) of the initial state of an arbitrary RM \( M \), and gives as output the code of the final state, if there is one. If \( M \) does not halt, \( U \) does not halt on input \( \langle M \rangle \).
7 The halting theorem

The main result of this part of the course is the

Theorem 8 There is no Register Machine \( H \) such that \( H \) takes as input an RM state encoding \( \langle M \rangle \) and halts with output 1 if \( M \) halts or output 0 if \( M \) does not halt.

We expand a little on the proof from the slides. The proof runs by contradiction: we suppose that such an \( H \) does exist, and show that this cannot be, by building another machine from \( H \) which says both ‘yes’ and ‘no’ to a particular question.

So, let \( H \) be 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.

We now construct a machine \( L = (P_L,R_0,\ldots) \), which takes a program code \( \langle P \rangle \) and terminates with 1 if \( H \) returns 0 on input \( \langle P,R_0 = \langle P \rangle \rangle \), and or goes into an infinite loop if \( H \) returns 1 on \( \langle P,R_0 = \langle P \rangle \rangle \). To construct \( L \), we just have to write code that, given \( \langle P \rangle \) in \( R_0 \), computes \( \langle P,R_0 = \langle P \rangle \rangle \), which by definition 6 is \( \langle \langle P \rangle,\langle \langle P \rangle \rangle \rangle \), copies that into \( R_0 \), and transfers control to the code of the machine \( H \). Then the ‘halting state’ of \( H \) (which, recall, is the empty instruction at the end) is replaced by code which changes \( R_0 \) (the output of \( H \)) to 1 if it is 0, and goes into a tight loop if it is 1. (You may find it helpful to draw a diagram for this construction.)

Thus, \( L \) takes a program, and runs the halting test on the program with itself as input – this is called (diagonalization – and loops iff the program halts, thereby reversing the halting behaviour of the program: \( L \) halts iff its input loops. The combination of diagonalization (or self-reference) and negation is the core of most theorems of this kind.

Now consider the result of running \( L \) 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.

Thus we conclude no such machine as \( H \) can exist, and the theorem is proved.

8 Turing Machines

The Halting Theorem was originally proved by Turing for his model of computation, now called Turing Machines. In this course, we are using Register Machines, as a more intuitive model of computation. Therefore I shall not provide here any further technical details of Turing Machines. The resources listed on the course webpage provide details if you want them.

It is not hard – that is, it is only a matter of writing large programs – to show that TMs and RMs are equivalent, in that we can write an RM program to simulate TMs, and a TM program to simulate RMs.

In fact, all plausible models of computation are equivalent in this sense. This is the Church–Turing Thesis. It’s not a theorem, because ‘plausible model of computation’ is not a formal notion, but nobody has ever come up with a counter-example. There are models of computation which can compute more than TMs can (and in particular can solve the Halting Problem of TMs), but they involve rather implausible tricks such as orbiting a suitably rotating black hole. See the Wikipedia article on ‘Hypercomputation’ for a summary.
9 Generalizing computability

We have now produced something that (we claim) is a universal model of computation, and also shown that there are things it cannot compute. Several questions now arise. Is the Halting Problem the only uncomputable problem? Are there different levels of uncomputability? Suppose we’re given a ‘real’ problem (not one about RMs!) – how can we tell whether it is computable or not?

If we have a real problem, then one way to show that it is computable is to write a program that solves it, and then prove (using suitable mathematical techniques specific to the problem, such as induction) that indeed the program always terminates. Another way is to translate it into another problem we already know about. For example, any problem about Turing Machines can be turned into a problem about Register Machines, so we don’t need to think independently about TM problems.

How do we show that a real problem is uncomputable? It’s not usually obvious how to carry out a diagonalization and negation argument on a real problem. However, it is often not too hard to see how to encode the Halting Problem (about RMs) into a particular example of our real problem. If we can do that, then we know the real problem is uncomputable in general – because if we could always compute the answer, then we could compute the answer to the encode Halting Problem.

To study these ideas, we need to introduce some more formal apparatus.

So far, we have thought about the computability of a problem with a ‘yes/no’ answer. Because we want to think about translating or encoding, it will be useful to think more generally about computable functions. Ultimately, we are always talking about integers, because that’s what RMs compute with, but since most mathematical or engineering concepts can easily be coded up as integers, we shall quite freely talk about computable functions on, for example, finite graphs, or rational numbers, or finite trees, and assume that the work of encoding and decoding happens behind the scenes.

For example, many problems concern finite graphs. A question we shall see later is ‘does $G$ have a $k$-clique?’ – that is, are there $k$ vertices of $G$ such that every one is connected to every other? In modern programming languages, we would define a suitable datatype for graphs. In RM programming, we have to code graphs as numbers. An obvious (but not necessarily the best) way to code a graph

$$G = (V = \{v_1, \ldots, v_n\}, E = \{e_1, \ldots, e_m\})$$

is as

$$\langle n, \langle\langle \text{src}(e_1), \text{dst}(e_1)\rangle_2, \ldots \langle\langle \text{src}(e_m), \text{dst}(e_m)\rangle_2\rangle_2 \rangle_2$$

where $\text{src}$, $\text{dst}$ indicate the source and target vertices of an edge.

**Definition 9** A (total) function $f : \mathbb{N} \to \mathbb{N}$ is **computable** if there is an RM/TM which computes $f$ (e.g. by taking $x$ in $R_0$ and leaving $f(x)$ in $R_0$) and always terminates. \(\triangleright\)

Historically, computable functions were often called ‘recursive functions’. This comes from another approach to computability theory in which we look at the ways of defining mathematical functions. With the advent of computers, programming languages, and the widespread use of recursion in programming, the term has become unhelpful: for example you can perfectly well write both computable and uncomputable functions in a language without recursion, and if you write a recursive (in the programming sense) function, it may or may not be computable. Most people now use ‘computable’, but when reading older literature, be aware that the term is used.
A ‘problem’ is just a special case of a function, as it can be viewed as a function where
the result is either 0 for ‘no’ or 1 for ‘yes’. Set membership is, of course, a predicate. We’ll
flip freely between notions. We’ll have a formal definition for this:

**Definition 10** A *decision problem* is a set $D$ (the *domain*) and a subset $Q$ (the *query*)
of $D$. The problem is ‘is $d \in Q$?’, or ‘is $Q(d) = 0$ or $1$?’. The problem is *computable* or *
decidable* iff the predicate $Q$ is computable.

Some example decision problems are:
- the domain $\mathbb{N}$ and the subset *Primes* of prime numbers;
- the domain $\mathbb{N} \times \mathbb{N}$ and the set $\{(p, p + 2) : p$ and $p + 2$ are both prime$\}$;
- the domain $RM$ of register machine (encodings), and the subset $H$ that halt;
- the domain of ASCII strings and the subset that are English sentences.

### 10 Oracles

Sometimes it is interesting (or technically useful) to be able to think about the question
‘what else could we compute if we *could* solve the Halting Problem?’, or more generally,
given some hard problem $Q$, ‘what could we compute if we *could* solve $Q$?’.

Oracles are a device to let us reason about such questions. An oracle is imagined to be
a black box that we can plug into our RM, which gives the answer to the hard problem $Q$
in a single computation step.

**Definition 11** Given a decision problem $(D, Q)$, an *oracle* for $Q$ is an additional RM
instruction $ORACLE_Q(i)$ which assumes that $R_i$ contains (an encoding of) some $d \in D$,
and sets $R_i$ to contain $Q(d)$.

Note that if $Q$ is itself decidable, then $ORACLE_Q(i)$ can be replaced by a call to a sub-
RM which computes $Q$ – thus a decidable oracle adds nothing. However, in the second
part of the course, when we consider computing with restricted resources, we shall again
use oracles for problems with more resource than normally allowed, and in that context
oracles for decidable problems make sense.