Introduction to Theoretical Computer Science

Julian Bradfield
jcb@inf.ed.ac.uk
IF–4.07

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
▶ 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 16 February and 23 March (end of weeks 5 and 9 – remember the gap week isn’t numbered).
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!

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.

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.
*Note:* there are no lectures in week 5. The last examinable lecture is (as currently planned) the Monday of week 9, but there are two further lectures offered.

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


ENIAC – 1945, Eckert & Mauchley

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/.
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:
   - \( \text{INC}(i) \): add 1 to \( R_i \), or
   - \( \text{DECJZ}(i, j) \): if \( R_i = 0 \) then goto \( I_j \), else subtract 1 from \( R_i \).
▶ The machine executes instructions in order, except when \( \text{DECJZ} \) causes a jump.

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

What is unrealistic about these machines?

Programming RM

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 \( \text{DECJZ} (-1,j) \)
▶ ‘clear \( R_i \)’
  0 \( \text{DECJZ} (i,2) \)
  1 \( \text{GOTO} 0 \)
▶ ‘copy \( R_i \) to \( R_j \)’ using \( R_{-2} \) as temp
  0 \( \text{CLEAR} R_j \)
  \( \text{loop1} : 2 \text{DECJZ} (i, \text{loop2}) \)
  3 \( \text{INC} (j) \)
  4 \( \text{INC} (-2) \)
  5 \( \text{GOTO} \text{loop1} \)
  \( \text{loop2} : 6 \text{DECJZ} (-2, \text{end}) \)
  7 \( \text{INC} (i) \)
  8 \( \text{GOTO} \text{loop2} \)
  \( \text{end} : 9 \)
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 $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

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

Given a pairing function $f$, write $(x, y)_2$ for $f(x, y)$. If $z = (x, y)_2$, 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 $(\ldots)_n : \mathbb{N}^n \to \mathbb{N}$. (Exercise: give two different ways.)

And thence to $(\ldots) : \mathbb{N}^* \to \mathbb{N}$

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.)

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:

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

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 $R_0$.

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.
**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.

**remarks**

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?
Computable functions

Much of what we say henceforth could be about any reasonable ‘domain’: graphs, integers, rationals, trees. We will use \( \mathbb{N} \) as the canonical domain, and rely on the (usually obvious) fact that any reasonable domain can be encoded into \( \mathbb{N} \).

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.

Note that predicates (e.g. ‘does the machine halt?’) are functions, viewing 0 as ‘no’ and 1 as ‘yes’.

You will also see the term ‘recursive function’. This is confusing, because it doesn’t mean ‘recursive’ in the sense you know, though there is a strong connection. This terminology is slowly going out of use. We may come to it . . .

Decision problems and oracles

A decision problem is a set \( D \) (domain) and a subset \( Q \) (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. For example:

- the domain \( \mathbb{N} \) and the subset \( \text{Primes} \);
- the domain \( \text{RM} \) of register machine (encodings), and the subset \( H \) that halt.

Given a decision problem \( (D, Q) \), an oracle for \( Q \) is a ‘magic’ RM instruction \( \text{ORACLE}_Q(i) \) which assumes that \( R_i \) contains (an encoding of) \( d \in D \), and sets \( R_i \) to contain \( Q(d) \).

If \( Q \) is itself decidable, \( \text{ORACLE}_Q(i) \) can be replaced by a call to an RM which computes \( Q \) – thus a decidable oracle adds nothing.

Proving (un)decidability

Is the halting problem the only natural (?) undecidable problem?

How can we show that a ‘real’ problem is decidable or undecidable?

To show a problem decidable: write a program to solve it, prove the program terminates. Alternatively, translate it to a problem you already know how to decide.

Why doesn’t this contradict the halting theorem?

To show a problem undecidable: prove that if you could solve it, you could also solve the halting problem.

Reductions

Reductions are a key technique. Simple concept: turn one problem into another. But there are subtleties.

A Turing transducer is an RM that takes an instance \( d \) of a problem \( (D, Q) \) in \( R_0 \) and halts with an instance \( d' = f(d) \) of \( (D', Q') \) in \( R_0 \). (Thus \( f \) is a computable function \( D \to D' \).)

A mapping reduction or many–one reduction from \( Q \) to \( Q' \) is a Turing transducer \( f \) as above such that \( d \in Q \) iff \( f(d) \in Q' \).

Equivalently, an m-reduction is a Turing transducer which after putting \( f(d) \) in \( R_0 \) then calls \( \text{ORACLE}_Q(0) \) and halts.

NOTE that there is no post-processing of the oracle’s answer!

Is this intuitively reasonable?

If you allow the transducer to call the oracle freely, you get a Turing reduction. These are also useful; but they’re more powerful, so don’t make such fine distinctions of computing power.
Showing undecidability

Reductions are the key (really the only) tool for showing undecidability, thus:

- Given \((D, Q)\), construct a reduction \(\text{Red}(H, Q)\) from \((\text{RM/TM}, H)\) to \((D, Q)\).
- Suppose \(Q\) is decidable. Then given \(M \in \text{RM}\), feed it to \(\text{Red}(H, Q)\) to decide \(M \in H\). But if \(Q\) is decidable, we can replace the oracle, and \(\text{Red}(H, Q)\) is just an ordinary RM. Hence \(H\) is decidable – contradiction. So \(Q\) must be undecidable.

Constructing \(\text{Red}\) is usually either very easy, or rather long and difficult!

For a relatively simple example of a long and complicated reduction, see https://hal.archives-ouvertes.fr/hal-00204625v2, a simplified proof by Nicolas Ollinger of the famous ‘tiling problem’ due to Hao Wang.

The Uniform Halting Problem

A simple example:

The Uniform Halting Problem \((UH)\) asks, given a machine \(M\), does \(M\) halt on all inputs?

‘Clearly’ at least as hard as \(H\), so must be undecidable. Proof:

- We need an m-reduction from \(H\) to \(UH\):
- given machine \(M\), input \(R\), build a machine \(M'\) which ignores its input and overwrites it with \(R\), then behaves as \(M\).
- Then \(M'\) halts on any input iff \(M\) halts on \(R\).

Check that this really is a transducer.

The Looping Problem

Let \(L\) be the subset of RMs (or TMs) that go into an infinite loop. Show that \(L\) is undecidable.

Since \(L\) is just the complement of \(H\), this seems equally easy: just flip the answer. But m-reductions can’t post-process the answer.

Can you devise an m-reduction from \(H\) to \(L\)?

No! You can’t.

Can you devise a Turing reduction from \(H\) to \(L\)?

When ‘yes’ is easier than ‘no’

The halting problem is not symmetrical:

- if \(M \in H\), then we can determine this: run \(M\), and when it halts, say ‘yes’;
- if \(M \notin H\), then we can’t determine this: run \(M\), but it never stops, and we can never say ‘no’.

Such problems are called semi-decidable.

The problem \(L\) is the opposite: we can determine 'no', but not ‘yes’. It is called co-semi-decidable.

There are two ways of exploiting this asymmetry – we’ll see them quickly now, and in more detail in the second section of the course.
**Enumeration**

It's convenient to talk of RMs that 'output an infinite list' – as in

\[ \texttt{i = 0; while ( true ) { print i++; }} \]

– which can be formalized in several ways. (**Exercise:** think of a few.)

Suppose \((\mathbb{N}, Q)\) is decidable. Then we can write a machine which outputs the set \(Q\):

- for each \(n \in \{0, 1, 2, \ldots\}\), compute \(Q(n)\) and output \(n\) iff \(Q(n)\):
- thus every \(n \in Q\) will eventually be output.

This is an enumeration of \(Q\), and we say that \(Q\) is computably enumerable (c.e. for short).

\(H\) is not decidable – but can we still enumerate it?

---

**Semi-decidability and enumerability**

The interleaving technique lets us see that

- Any semi-decidable problem is also computably enumerable

**Exercise:** Show that the converse is true:
- Any c.e. problem is semi-decidable.

Now we've shown that semi-decidability is the same as c.e.-ness.

Note that if \((D, Q)\) is semi-decidable, then \((D, D \setminus Q)\) is co-semi-decidable.

---

**Enumeration by interleaving**

Observe that the set of valid register machine encodings is decidable: given \(n\), we can check whether \(n\) is \(\texttt{⌜M⌝}\) for a syntactically valid machine \(M\).

Therefore we can enumerate \(\texttt{⌜M_0⌝,⌜M_1⌝,\ldots}\

Now consider the following pseudo-code:

\[
\text{machineList} := \langle \rangle \quad \text{sequence storing a list of machine (code)}
\]

\[
\text{for (i:=0; true; i++):}
\]

\[
\begin{align*}
\text{add } \texttt{⌜M_i⌝ to machineList} \\
\text{foreach } \texttt{⌜M⌝ in machineList:}
\end{align*}
\]

\[
\begin{align*}
\text{run } M \text{ for one step and update in machineList} \\
\text{if } M \text{ has halted:}
\end{align*}
\]

\[
\begin{align*}
\text{output } \texttt{⌜M⌝} \\
\text{delete } \texttt{⌜M⌝ from machineList}
\end{align*}
\]

This program outputs \(H\): every halting machine is eventually output.

\(H\) is computably enumerable.

---

**Back to Looping**

Suppose that \((D, Q)\) is both semi-decidable and co-semi-decidable.

- So \(Q\) and \(D \setminus Q\) are both semi-decidable.
- So run a semi-deciding machine for \(Q\) in parallel with a semi-decider for \(D \setminus Q\), using interleaving.
- Whatever instance \(d\) we look at, one of the two will halt – so stop then.

So \((D, Q)\) is decidable.

Now we know that \(L\) cannot be semi-decidable, as it’s the complement of semi-decidable \(H\).

**Exercise:** When I say the complement of \(L\) is \(H\), what is the domain \(D\)? Do I need to take care?)
Uniform Halting again

We showed, easily, that \( UH \) was undecidable. Is it semi-decidable?
We used reductions from \( H \) to show un-decidability. Do reductions from \( L \) show un-semi-decidability? Yes!
Suppose \( f \) reduces \( Y \) to \( X \), and suppose that \( X \) is semi-decidable by machine \( M_X \). Then since \( Y(y) = X(f(y)) \), if \( Y(y) = 1 \) we can translate \( y \) to \( f(y) \), run \( M_X \), and get 1. If \( Y(y) = 0 \), then \( M_X \) applied to \( f(y) \) either gives 0, or doesn’t halt.
So if \( X \) is semi-decidable, so is \( Y \); or if \( Y \) is not semi-decidable, then \( X \) can’t be.

Reducing \( L \) to \( UH \)

We want a transducer \( f(M, R) \) such that \( f(M, R) \) halts on all inputs iff \( M \) loops on \( R \). Not obvious … we could ignore the inputs to \( f(M) \), but how do we make it stop?
By a clever trick: \( f(M, R) \) will measure how long \( M \) runs for, with a timeout as its input.
- Let \( M' = f(M, R) \) be a machine which takes a number \( n \) as input, and then simulates \( M \) on \( R \) for (at most) \( n \) steps.
- If \( M \) halts on \( R \) before \( n \) steps, then \( M' \) goes into a loop;
- while if \( M \) hasn’t halted, \( M' \) just halts.
- Hence if \( M \) loops on \( R \), \( M' \) halts on all inputs \( n \); and if \( M \) halts, then \( M' \) loops on all sufficiently large \( n \), and so doesn’t halt on all inputs.

So \( UH \) is not …

The reduction from \( L \) shows that \( UH \) is not semi-decidable.
Is \( UH \) co-semi-decidable (like \( L \))? No! The (trivial) reduction from \( H \) to \( UH \) also shows that \( UH \) is not co-semi-decidable.

\[ \begin{align*}
\Sigma^0_1 \text{ semi-decidable} & \quad \Delta^0_1 \text{ decidable} \\
H & \quad \text{semi-decidable} \quad \Pi^0_1 \\
L & \quad \text{co-semi-decidable} \quad \Pi^0_1 \\
UH & \quad \text{not semi-decidable} \quad \text{not co-semi-decidable}
\end{align*} \]

\( H \) has from ‘\( \exists n.M \) halts after \( n \)’; \( L \) has form ‘\( \forall n.M \) doesn’t halt after \( n \)’; \( UH \) has form ‘\( \forall R.\exists n.M \) halts after \( n \) on \( R \)’. Really the diagram should be …

Harder and harder

Suppose we consider \( RM^H \), the class of register machines with an oracle for \( H \).
We can easily adapt our universal machine to produce a universal\(^H \) machine for this class.
So we can apply diagonalization, and show that:
- There is no \( RM^H \) machine that computes the halting problem for \( RM^H \)
We can keep doing this, and generate harder and harder problems – the arithmetical hierarchy. Sadly, we don’t have (much) time to explore this fascinating topic.
Complexity of algorithms and problems
Recall that in Inf2B (and in ADS if you did it) we calculated the running cost of algorithms: typically as a function \(f(n)\) of the size \(n\) of the input.
Recall big-O notation: \(f \in O(g)\) means that \(f\) is asymptotically bounded above by \(g\), or formally
\[
\exists c, n_0. \forall n > n_0. f(n) \leq c \cdot g(n)
\]
Similarly \(f \in \Omega(g)\) means \(f\) is bounded below by \(g\)
\[
\exists c, n_0. \forall n > n_0. f(n) \geq c \cdot g(n)
\]
Recall that for some problems, we could show that any algorithm must take at least a certain time. E.g. any sorting algorithm must be \(\Omega(n \lg n)\), and we have an \(O(n \lg n)\) algorithm.

Variations in models of computing
In Inf2/ADS, there was the question of what counts as a step of computation.
E.g. in sorting, we counted the number of comparisons – and assumed that it costs as much to compare a small number as a big number.
Reasonable, for numbers that fit in a machine word!
What about the cost of control flow etc.?
Do we care about whether we’re using Turing machines, register machines, random access machines, or whatever?
On a register machine, adding two numbers is an \(O(n)\) operation (because of the decrement/increment loop). But if we add addition as a primitive, it’s \(O(1)\). On a Turing machine, it’s \(O(\lg n)\) (work in binary, and do primary school long addition).
It would be nice to ignore these differences.

What counts as seriously different?
Can we separate easy problems from hard problems?
If a problem is \(O(n)\) on some model, it’s surely easy on any model.
Actually, \(O(n)\) is not ‘easy’ when \(n\) is a petabyte. There’s a whole field of ‘sub-linear’ algorithms.
If a problem is \(\Omega(2^n)\) on some (non-unary) model, it’s surely hard on any model.
Actually, there are problems that are much worse than \(\Omega(2^n)\), but still quite solvable for real examples. We’ll see one later.
What about something that is \(O(n^{10})\) or \(\Omega(n^{10})\)?
An \(n^{10}\) problem is probably practically insoluble – but maybe there’s a trick or a fancy new model that makes it \(O(n^2)\)?
There’s also that \(c \ldots \) if \(f(n) \geq 10^{100} \lg n\), that’s not very tractable. But this doesn’t often happen.

Time to extend our RMs a bit
Our very basic RMs essentially work in unary: that’s why addition is \(O(n)\) instead of \(O(\lg n)\).
This is too big a slow-down to ignore – it’s exponential in the size (number of bits = \(\lg n\)) of the input!.
So from now on we assume instructions \(\text{ADD}(i, j)\) and \(\text{SUB}(i, j)\) which (add/truncated subtract) \(R_j\) from \(R_i\), putting the result in \(R_i\). (This is enough to do everything else.)
Now our machines are too fast, because we’re adding in time 1, instead of in time \(O(\lg n) = O(\text{bits})\). For our purposes now, this is just a linear factor, so doesn’t matter. If you prefer, you can refine the machine to have registers implemented as bit sequences, with bit manipulation instructions.
P – the class of polynomial-time problems
A problem (decision problem or function) or algorithm is in P or \( P \text{Time} \) if it can be computed in time \( f(n) \in O(n^k) \) for some \( k \).
The class \( P \) is robust: any reasonable change of model doesn’t change it, and moreover any reasonable translation of one problem into another preserves membership in \( P \).
Problems in \( P \) are called tractable.
This is somewhat bogus (see previous slide).
Nonetheless, a problem in \( P \) is generally easy-ish in practice, and a problem not in \( P \) is generally hard in practice.
Most of the algorithms you’ve seen are in \( P \).
Note that if a problem is not in \( P \), it’s \( \Omega(n^k) \) for every \( k \) (and so \( \omega(n^k) \) for every \( k \)). E.g. \( 2^n \), or \( 2^{\sqrt{n}} \).

Outside P
What does it mean to say that a problem cannot be computed by a polynomial algorithm? What is an algorithm?
A polynomially-bounded RM is an RM together with a polynomial (wlog \( n^k \) for some \( k \)), such that if the size of the input (e.g. the number of bits of the number in \( R_0 \)) is \( n_0 \), then the machine halts after executing \( n_0^k \) instructions.
Then formally a problem \( Q \) is in \( P \) iff it can be computed by some polynomially-bounded RM.
This avoids the problem of having to analyse the running time of the machine (which we know is impossible in general!).
To show that \( Q \in P \), we therefore have to show that every polynomially-bounded RM claiming to solve \( Q \) will time out on some input.
This is not all that easy . . .

Polynomial-time reductions
Just as with computability, reductions are the key tool for comparing the complexity of problems.
A \( \text{PTime} \) reduction or polynomial reduction from \((D, Q)\) to \((D’, Q’)\) is:
- a \( \text{PTime} \)-computable function \( r : D \to D’ \), such that
- \( d \in Q \iff r(d) \in Q’ \)
We say \( Q \leq^p Q’ \)
Clearly, if \( Q’ \in P \) and \( Q \leq^p Q’ \), then also \( Q \in P \).
Exercise: If we replace ‘\( \text{PTime} \)-computable’ by ‘computable’, do we get the notion of many–one reduction we used before, or the notion of Turing reduction? Does it matter?
For function problems, it makes sense to use a Turing reduction. Exercise: Express this in terms of oracles.

Some apparently intractable problems
Practical problems that are hard to solve typically seem to need exploring many (exponentially many) possible answers.
The Hamiltonian Path Problem takes a graph \( G = (V, E) \), and asks if there is a Hamiltonian path, i.e. a path that visits every vertex exactly once.
Naively, the only way to solve it is to explore all paths, and there could be up to \(|V|!\) of them. Factorial grows slightly faster than exponential, so this is bad . . .
Timetabling is the following: given students taking exams, and timetable slots for exams, is it possible to schedule the exams so that there are no clashes? It also apparently requires looking at exponentially many possible assignments. (That’s why Registry starts timetabling exams 9 weeks in advance. . .)
Checking vs solving

Both HPP and Timetabling have the property that they are easy to check: if you have a claimed solution, it’s quick to verify that it is a solution. This is a feature of many practical intractable problems. Can we study and explore this property?

Non-deterministic register machines

In Inf2A, you saw non-deterministic finite automata: multiple possible transitions from each (state,input) pair. These were equivalent to deterministic automata, albeit with an exponential blow-up in states. We can do the same with RMs (or TMs). For RMs, probably easiest to add a new instruction \texttt{maybe(j)}, which non-deterministically either does nothing or jumps to \texttt{j}:

\begin{verbatim}
CLEAR \texttt{R0}
\texttt{beg : maybe (end)}
INC \texttt{(0)}
\texttt{goto \texttt{beg}}
\texttt{end :}
\end{verbatim}

puts a non-deterministically chosen number in \texttt{R0}.

There is a path which just loops for ever – this doesn’t matter. Usually we want to choose from a finite set, so not an issue anyway.

There is no notion of probability here. Probabilistic RMs (e.g. \texttt{maybe} chooses with equal prob.) are another extension.

An NRM accepts if some run (sequence of instructions through the choices) halts and accepts. `run accepts' could mean: halts; halts with 1 in \texttt{R0}; or whatever.

\textbf{Exercise:} explore a few possibilities – does it matter?

Thus infinite runs are ignored.

I sometimes prefer to think of \texttt{maybe} as \texttt{fork}: the machine forks a copy of itself which takes the jump. If any copy accepts, it signals the OS, which kills off all the others.

How does this work for problems with multiple correct answers?

We can use an RM to simulate all the runs of an NRM using the interleaving technique. Thus:

\begin{itemize}
  \item NRM accepts if \texttt{fork/}explores \texttt{2^O(n)} possibilities.
  \item \texttt{HPP is in NP: guess the Hamiltonian path (if there is one), and check it (in linear time).}
  \item \texttt{Timetabling is in NP: guess a timetable, and check it for clashes (in maybe O(n^2) time?).}
\end{itemize}

Exercise: Read the details of the TM version in Sipser.

NRMs are, however, potentially exponentially faster than RMs: in time \texttt{n}, an NRM can fork/explore \texttt{2^O(n)} possibilities.

We say \(Q \in \text{NP}\) if there is a polynomially-bounded NRM that computes \(Q\).

\begin{itemize}
  \item \texttt{HPP is in NP: guess the Hamiltonian path (if there is one), and check it (in linear time).}
  \item \texttt{Timetabling is in NP: guess a timetable, and check it for clashes (in maybe O(n^2) time?).}
\end{itemize}

Is this just theory? We can’t do non-determinism in practice . . . can we? Quantum computers can achieve a similar effect: an \(n\)-qubit computer computes on all \(2^n\) values simultaneously. But it’s hard to get many qubits; and there are subtleties – not every NP algorithm is quantum-computable (as far as we know).
Is NP all we need?
Is every apparently exponential problem really an NP problem? No!
▶ Consider the problem of determining whether an exponentially
bounded RM accepts.
▶ There is no shortcut, and nothing to guess. You just have to run it
and see.
To say nothing of doubly-exponential, non-elementary, etc. (See later.)

Are all NP problems equally hard?
What does this mean? How do we measure hardness?
We’ll measure hardness relative to $\leq^P$.
▶ $Q$ is easier than (or equally hard as) $Q'$ iff $Q \leq^P Q'$.
▶ We say $Q$ is NP-hard iff $Q \geq^P Q'$ for every $Q' \in \text{NP}$.
Truly exponential problems are NP-hard, but that’s boring.
▶ We say $Q$ is NP-complete iff $Q$ is NP-hard and $Q \in \text{NP}$.
▶ Obviously if $Q$ is NP-complete, and $\text{NP} \ni Q' \geq^P Q$, then $Q'$ is also
NP-complete.
Do NP-complete problems necessarily exist?
They do exist, and there are many, including HPP and Timetabling. In
fact, almost all NP-problems encountered in practice are either in P, or
NP-complete.

The Cook–Levin theorem
states that a particular NP problem, SAT, is NP-complete.
The theorem is usually proved for TMs; we shall do it for RM.
The concept is simple, but there are details. Most details are left for you.
Why ‘Cook–Levin’? The notion of NP-completeness, and the theorem, were
due to Stephen Cook (and partly Richard Karp) – in the West. But as with
many major mathematical results of the mid-20th century, they were discovered
independently in the Soviet Union, by Leonid Levin. Since the fall of the Iron
Curtain made Soviet maths more accessible, we try to attribute results to both
discoverers.

SAT
SAT is a simple problem about boolean logic:
Given a boolean formula $\phi$ over a set of boolean variables $X_i$, is there an
assignment of values to the $X_i$ which satisfies $\phi$ (i.e. makes $\phi$ true)?
Equivalently, is $\phi$ non-contradictory?
▶ $(A \lor B) \land (\neg B \lor C) \land (A \lor C)$ is satisfiable, e.g. by making $A$ and $C$
true.
▶ $(A \land B) \land (\neg B \land C) \land (A \land C)$ is not satisfiable.
The size of a SAT problem is simply the number of symbols in $\phi$.
SAT is obviously in NP: just guess an assignment and check it.
It’s also apparently exponential in reality: no obvious way to avoid
checking all possible assignments (the truth table method).
SAT is NP-complete
Suppose \((D, Q) \in \text{NP}\). We shall construct a reduction \(Q \leq^p \text{SAT}\).

Outline of proof:
Given an instance \(d \in D\), design a boolean formula \(\phi_d\) which can be satisfied if its variables describe the successful executions of an NRM checking \(Q\). The machine can be polynomially bounded, so the size of \(\phi_d\) is polynomial in the size of \(d\).

The rest is detail.

Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>How many?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C_{tj})</td>
<td>program counter at step (t) is on (I_j)</td>
<td>(s \cdot n)</td>
</tr>
<tr>
<td>(R_{tik})</td>
<td>(k)th bit of (R_i) at step (t)</td>
<td>(s \cdot m \cdot 2s)</td>
</tr>
</tbody>
</table>

Formulas

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>How big?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\chi_{\text{one}})</td>
<td>prog cntr in one place</td>
<td>(s \cdot n^2)</td>
</tr>
<tr>
<td>(\chi_t)</td>
<td>step (t + 1) follows from step (t)</td>
<td>(s^2 \cdot m)</td>
</tr>
<tr>
<td>(\rho_{\text{init}})</td>
<td>initial register values</td>
<td>(m \cdot n)</td>
</tr>
<tr>
<td>(\alpha)</td>
<td>machine accepts</td>
<td>(s)</td>
</tr>
</tbody>
</table>

The formula \(\chi_{\text{one}}\) ∧ \(\rho_{\text{init}}\) ∧ \(\chi_t\) ∧ \(\alpha\) is then satisfied by assignments corresponding to valid executions of the machine.

The \(\chi_{\text{one}}\) formula is easy:

\[
\bigwedge_t \bigvee_j \left( C_{tj} \land \bigvee_{j' \neq j} \neg C_{tj'} \right)
\]

more details

Coding \(\chi_t\) is a bit tedious . . . step \(t + 1\) follows from step \(t\) if the control flow is correct and if the registers have changed correctly.

For control flow: the formula is \(\bigvee_j (C_{tj} \land \nu_{ij})\), where \(\nu_{ij}\) is:

- \(C_{t+1,j+1}\) if \(i_j\) is INC, ADD OF SUB
- \(C_{t+1,j+1} \lor C_{t+1,j'}\) if \(i_j\) is MAYBE\((j')\)
- \((\bigvee_k R_{tik}) \land C_{t+1,j+1} \lor ((\bigwedge_k \neg R_{tik}) \land C_{t+1,j'})\) if \(i_j\) is DEC\((i,j')\)

To code the register changes, we need to express the addition of \((2s)\)-bit numbers by boolean operations on the bits. See your hardware course!

Exercise: write the formula \(\rho_{\text{add}}^s\) which says that at step \(t + 1\), the value of \(R_i\) is the sum of the values of \(R_i\) and \(R_{ix}\) at step \(t\). (Answer in the notes.)

Despite the very tedious nature of these formulas, we can see that the whole formula is \(O(s^4)\).

And so the theorem is proved.

Detail

We have an instance \(d \in D\).
Since \((D, Q) \in \text{NP}\), there is a polynomially bounded NRM \(M = (R_0, \ldots, R_{m-1}, l_0 \ldots l_{n-1})\) that computes \(Q\). Let \(p(d)\) be the polynomial bound, and let \(s = p(|d|)\). (\(s\) \(\text{‘number of steps’}\)) (Note: the empty instruction \(l_n\) is the halting state.)

So \(M\) takes at most \(s\) steps.

How big can the values in the registers get? Worst case: start with \(2^d\), and execute \(s\) ADD(0, 0) instructions, giving \(2^d + s\). I.e. at most \(|d| + s\) bits. Assume \(wlog\) \(s \geq |d|\), so at most \(2s\) bits.

We now define a large, but \(O(\text{poly}(s))\), number of variables, with intended interpretations.

‘Intended’ because we will design \(\phi_d\) so that these interpretations do what we want.

Exercise: write the formula \(\rho_{\text{add}}^s\) which says that at step \(t + 1\), the value of \(R_i\) is the sum of the values of \(R_i\) and \(R_{ix}\) at step \(t\). (Answer in the notes.)

Despite the very tedious nature of these formulas, we can see that the whole formula is \(O(s^4)\).

And so the theorem is proved.
More NP-complete problems
There are many hundreds of natural problems that are NP-complete. Look at the Wikipedia article ‘List of NP-complete problems’ for a range of examples, or Sipser for more details on a smaller range.
Starting from SAT, we use reductions to build a library of NP-complete problems, for use in tackling new potential problems.
Sometimes, the reductions from SAT (or other known problem) require considerable ingenuity. E.g., showing NP-completeness of HPP is quite tricky.
We’ll do a couple of examples.

3SAT
There are special forms of SAT that turn out to be particularly handy. A literal is either a propositional variable \( P \) or the negation \( \neg P \) of one. \( \phi \) is in conjunctive normal form (CNF) if it is of the form \( \bigwedge_i \bigvee_j p_{ij} \) where each \( p_{ij} \) is a literal.
\( \phi \) is in \( k \)-CNF if each clause \( \bigvee_j p_{ij} \) has at most \( k \) literals.
3SAT is the problem of whether a satisfying assignment exists for a formula in 3-CNF.
The reduction from unrestricted SAT to 3-SAT is also a bit tricky, mostly because normal boolean logic conversion to CNF introduces exponential blowup. The Tseitin encoding is used to produce a not equivalent but equisatisfiable CNF formula. (See Note 3.)
We are paying a price here for having done Cook–Levin with RMs. The original version with TMs manages to produce a CNF formula describing machine executions. In fact, SAT is often defined to mean CNF-SAT.

CLIQUE
Given a graph \( G = (V, E) \) and a number \( k \), a \( k \)-clique is a \( k \)-sized subset \( C \) of \( V \), such that every vertex in \( C \) has an edge to every other. (\( C \) forms a complete subgraph.) The CLIQUE problem is to decide whether \( G \) has a \( k \)-clique.
The problem has applications in chemistry and biology (and of course sociology).
It is NP-complete, by the following reduction from 3SAT:
Let \( \phi = \bigwedge_{1 \leq i \leq k} (x_{i1} \lor x_{i2} \lor x_{i3}) \) be an instance of 3SAT, so each \( x_{ij} \) is a literal. Construct a graph thus: each \( x_{ij} \) is a vertex. Make an edge between \( x_{ij} \) and \( x_{i'j'} \) just in case \( i \neq i' \) and \( x_{i'j'} \) is not the negation of \( x_{ij} \). (I.e., we connect literals in different clauses just when they are not inconsistent with each other.)
Since the vertices in one clause are disconnected, finding a \( k \)-clique amounts to finding one literal for each clause, such that they are all consistent – and so represent a satisfying assignment. Conversely, any satisfying assignment generates a \( k \)-clique.

P \( \equiv \) NP
We have said that NP-COMPLETE problems are effectively exponential. But do we know this? Is it possible that we’re just stupid, and there’s a PTIME algorithm for SAT?
We don’t know.
It is possible that NP = P. If you find a polynomial algorithm for SAT or any other NP-complete problem . . . hire bodyguards, because most web/banking security depends on such problems being hard.
Solving the problem is worth a million US dollars: it is one of the seven problems chosen by the Clay Institute for their Millennium Prizes.
Many of the results in complexity theory are therefore conditional: ‘if \( P \neq NP \), then very difficult theorem’.
E.g. if \( P \neq NP \), then there are problems that are neither P nor NP-complete. There are very few candidates: best known is the Graph Isomorphism Problem.
Handling NP

As far as we know, NP problems are just hard: need exponential search, so $O(\text{poly}(n) \cdot 2^n)$. So how do we solve them in practice? Huge industry ...

Randomized algorithms are often useful. Allow algorithms to toss a coin. Surprisingly one can get randomized algorithms that solve e.g. 3SAT in time $O(\text{poly}(n) \cdot (4/3)^n)$ (Why is this useful? $2^{100} \approx 10^{31}$, while $1.33^{100} \approx 10^{12}$) Catch: (really) small probability of error!

In many special classes (e.g. sparse graphs, or almost-complete graphs), heuristics lead to fast results.
See http://satcompetition.org/ for the state of the art.

Beyond NP

So what do we know about hard problems?
As noted earlier, $\text{ExpTime}$ is strictly harder than $\text{P}$ or NP: a polynomially bounded machine simply doesn’t have time or memory to explore $2^n$ steps. Formal proof via the Time Hierarchy Theorem.

Similarly $\text{2-ExpTime} \supseteq \text{ExpTime}$.

Also applies to $N$ versions.

The polynomial hierarchy

Recall the notion of oracle. Suppose we have an oracle for NP problems. What then can we compute?

The polynomial hierarchy is defined thus:

- Let $\Delta^p_0 = \Sigma^p_0 = \Pi^p_0 = \text{P}$.
- Let $\Delta^p_{n+1} = \text{P(Poly)}$, $\Sigma^p_{n+1} = \text{NP(Poly)}$, $\Pi^p_{n+1} = \text{co-NP(Poly)}$.

where co-X is problems whose negation is in X.

So $\Sigma^p_1 = \text{NP(Poly)} = \text{NP}$, and $\Sigma^p_2 = \text{NP}^{\text{NP}}$. Why is $\text{NP}^{\text{NP}}$ not just NP?

This hierarchy could collapse at any level – generalization of $\text{P} = \text{NP}$.

We don’t know.

The whole hierarchy is inside $\text{PSPACE}$.

Space-bounded machines

An RM/TM is $f(n)$-space-bounded if it may use only $f(input size)$ space. For TMs, space means cells on tape; for RMs, number of bits in registers. $\text{PSPACE}$ is the class of problems solvable by polynomially-space-bounded machines.

The following are obvious (Exercise: why?):

- $\text{PSPACE} \supseteq \text{PTime}$
- $\text{PSPACE} \supseteq \text{NPTime}$
- $\text{PSPACE} \subseteq \text{ExpTime}$
Lambda-Calculus
While Turing was working with machines, Alonzo Church was computing differently – with the precursor of Haskell!
You saw Haskell as a programming language for a computer: Church saw it as the computer.
Computation with lambda-calculus works by reduction (syntactic manipulation) of expressions or terms.

Lambda-calculus syntax
The terms of lambda-calculus are:

▶ There is a set of variables $x, y, \ldots$. Every variable is a term.
▶ If $x$ is a variable and $t$ is a term, then $(\lambda x. t)$ is a term. (abstraction)
▶ If $s$ and $t$ are terms, then $(st)$ is a term. (application)
▶ Nothing else is a term.

To avoid writing Lots of Irritating Spurious Parentheses, we adopt the conventions:

▶ Omit outermost parentheses. (E.g. $\lambda x. t$)
▶ Application associates to the left: $stu$ means $(st)u$.
▶ the body of an abstraction extends as far to the right as possible: $\lambda x. sly. xty$ means $\lambda x. (s(ly.((xt)y)))$

Free and bound variables; $\alpha$-conversion
If a variable $x$ occurs inside a term $t$, the occurrence is free iff $x$ is not inside a subterm $\lambda x. t'$ of $t$.
If $x$ is free in $t'$, it is bound in $\lambda x. t'$.
This is just like variable binding in all civilized programming languages.

Exercise: Write a formal definition of ‘free’ and ‘bound’ by induction on the structure of terms.

The terms $\lambda x. x$ and $\lambda y. y$ are not interestingly different.
Changing the name of a bound variable is $\alpha$-conversion (historical reasons. . . ).

Care is needed: can’t just change $x$ to $y$ in $\lambda x. ly. xy$, or in $\lambda x. yx$.

Exercise: Write a precise and safe definition of $\alpha$-conversion.
We write $s \alpha\rightarrow t$ if $s$ can be $\alpha$-converted to $t$.

Intuition
Variables are . . . variables. Abstraction makes anonymous functions: $\lambda x. t$ is function that takes an argument ($s$, say), binds $s$ to the variable $x$, and evaluates (?) $t$.
Application is application of a function to an argument.
Examples:

▶ $\lambda x. x$ is the identity function
▶ $(\lambda x. x)(\lambda x. x)$ is the identity function applied to itself – which is (will turn out to be) the identity function
▶ $\lambda x. \lambda y. wxy$ is a function that takes an argument $x$, and gives back a function that takes an argument $y$ and applies $w$ to $x$ giving a result that is applied to $y$. A curried function of two arguments.

This should all look completely familiar from Haskell – except there are no types here! Also no integers, booleans, data types, pattern matching, . . .
Evaluating terms: $\beta$-reduction

The main computation is done by $\beta$-reduction (historical reasons...), which 'evaluates' a function application:

$$(\lambda x.t)(s) \xrightarrow{\beta} t[s/x]$$

where $t[s/x]$ is the result of (appropriately) substituting $s$ for every free occurrence of $x$ in $t$.

Examples:

$$(\lambda x.x)(\lambda y.y) \xrightarrow{\beta} \lambda y.y$$

$$(\lambda x.xx)(\lambda y.y)w \xrightarrow{\beta} (\lambda y.y)(\lambda y.y)w$$

$$(\lambda y.y)w \xrightarrow{\beta} w$$

Variable capture and substitution

The lambda calculus is purely text (symbol) manipulation – naive substitution can go wrong:

$$(\lambda x.\lambda y.xy)yw \rightarrow (\lambda y.yy)w \rightarrow ww$$

because the free $y$ was captured by $\lambda y$.

So we define $\xrightarrow{\beta}$ to do the right thing, by (e.g.) renaming bound variables to avoid capture:

$$(\lambda x.\lambda y.xy)yw \xrightarrow{\beta} (\lambda y'.yy')w \xrightarrow{\beta} yw$$

Surprisingly tricky to get right. See Pierce §5.3 for details.

Eta conversion

Sometimes it’s useful to wrap a function up in another abstraction (e.g. to delay its evaluation). $\eta$-conversion allows us to do this: provide $x$ is not free in $f$, then

$$(\lambda x.fx) \xrightarrow{\eta} f$$

Evaluation order

Does it matter what order we evaluate things in?

$$(\lambda z.w)((\lambda x.xx)(\lambda x.xx)) \xrightarrow{\beta} w$$

or

$$(\lambda z.w)((\lambda x.xx)(\lambda x.xx)) \xrightarrow{\beta} (\lambda z.w)((\lambda x.xx)(\lambda x.xx))$$

$$(\lambda z.w)((\lambda x.xx)(\lambda x.xx)) \xrightarrow{\beta} (\lambda z.w)((\lambda x.xx)(\lambda x.xx))$$

...'

'Call by name' (nearly but not quite what Haskell does) is the first: reduce the outer left redex first, and don’t go inside lambda abstractions.

'Call by value' is the second: reduce function arguments before reducing the application.

For others, see Pierce §5.1.
Adding stuff
So far, this doesn’t look like much of a computer.
We could imagine adding basic numerals 0, 1, 2, ..., and arithmetic operations with rules like
\[ +n_1n_2 \xrightarrow{\beta} n \text{ where } n = n_1 + n_2 \]
and booleans true, false and conditionals with rules like
\[ \text{if } bst \xrightarrow{\beta} s \text{ if } b \text{ is true} \]
\[ \text{if } b \text{ is false} \]
but we still need some way to achieve iteration or looping.
It’s not actually necessary to add numerical primitives: we can define \( \lambda \)-terms to represent them. Look up Church numerals in the textbook.

Recursion in lambda calculus
Haskellers will expect that recursion is the trick. But without names for functions it’s ... tricky.
Example: factorial function is
\[
(\lambda F.(\lambda X.F(XX))(\lambda X.F(XX)))(\lambda f.(\lambda x.(\text{if } (= 0 x) 1 (\times x(f(-x 1)))))
\]
If you can understand that ... The \( \lambda f \ldots \) is basically factorial, but we have to find a way for the embedded \( f \) to refer to itself.
Exercise: work through the evaluation applied to 3. (Use call by name.)
At this point, you may wish to find a lambda-calculus evaluator. (There are many. Explore!)
Exercise: can you do this in Haskell? If not, why not?

Putting \( G \overset{\text{def}}{=} \lambda f.(\lambda x.(\text{if } (= 0 x) 1 (\times x(f(-x 1))))) \), and evaluating \( Y G 3 \), say, we get:
\[
Y G 3 \xrightarrow{\beta} G (Y G) 3 \xrightarrow{\beta} (\lambda x.(\text{if } (= 0 x) 1 (\times x(Y G(-3 1)))) 3 \xrightarrow{\beta} \text{if } (= 0 3) 1 (\times 3 (Y G(-3 1))) \xrightarrow{\beta} \times 3 (Y G(-3 1)) \xrightarrow{\beta} 6
\]

\( Y ? \)
The magic term that made \( \lambda f \ldots \) recursive was
\[
Y \overset{\text{def}}{=} \lambda F.(\lambda X.F(XX))(\lambda X.F(XX))
\]
Why does it work?
Suppose we give \( Y \) an argument \( G \), a function that takes two arguments \( f \) and \( x \). Then, in call-by-name,
\[
YG \xrightarrow{\beta} (\lambda X.G(XX))(\lambda X.G(XX)) \xrightarrow{\beta} G((\lambda X.G(XX))(\lambda X.G(XX)))
\]
Observe that the last term is just (one reduction of) \( G(Y G) \).
Now if \( G \) itself applies its first argument (\( Y G \) here) to its second argument, \( Y G \) will get expanded to \( G(Y G) \) again. If doesn’t use its first argument, the expansion stops.
Types
In λ-calculus, we can write crazy things like
\((\lambda x. xx)(0)\)
or
\((+(\lambda x. xx)42)\)
Simply typed λ-calculus adds simple (!) types to the basic λ-calculus syntax, to prevent such nonsense.

- Fix some set of base types, e.g. nat, bool, for the constants.
- If σ, τ are types, then σ → τ is a type, the function type of abstractions that take an argument of type σ and return a value of type τ. **Convention:** → associates to the right, so σ → τ → ν means σ → (τ → ν).
- The syntax of abstractions is changed to (λx:σ.t), giving the type of the argument.

Note that σ, τ are meta-variables, not symbols in the language itself.

Proofs for types
It’s common to present type inferences as proof trees, as with proofs in propositional or predicate logic.

**We assume** that all formulae are α-converted to avoid clashes of bound variables.

\(Γ ⊢ t : τ\) is a judgement that \(t\) has type \(τ\) under the set of assumptions \(Γ\). The following rules (read top to bottom) prove judgements:

- \(Γ ⊢ c : τ\) for each constant \(c\) of base type \(τ\).
- \(Γ, x : σ ⊢ t : τ\)
- \(Γ ⊢ λx:σ.t : σ → τ\)
- \(Γ ⊢ s : σ\)
- \(Γ ⊢ t : σ → τ\)
- \(Γ ⊢ ts : τ\)

Then \(t : τ\) iff \(⊢ t : τ\) is provable.

Typing
A legitimate simply typed λ-calculus term must be well typed. This is determined by inferring types.

For a term \(t\) and type \(τ\), we write \(t : τ\) for ‘\(t\) is well typed with type \(τ\)’.

- Constants (individuals or functions) are well typed with their assigned type.
- If \(t : τ\) under the assumption that \(x : σ\), then (λx:σ.t) : σ → τ.
- If \(t : σ → τ\) and \(s : σ\), then (ts) : τ.

Examples:
- A function that takes a function \(f\) on integers and returns a function that applies \(f\) twice:
  \((λf:nat → nat.λx:nat.f(fx)) : (nat → nat) → (nat → nat)\)

**Exercise:** Work it through.
- \(λx:nat.xx\) cannot be well typed.

Example: to show that \(λx:nat.(+ 1 x) : nat → nat:\)

\[\begin{align*}
  x : nat &\mid + : nat → (nat → nat) \\
  x : nat &\mid 1 : nat \\
  x : nat &\mid ∈ \{ x : nat\} \\
  x : nat &\mid (+ 1) : nat → nat \\
  x : nat &\mid x : nat \\
  x : nat &\mid (+ (1 + x)) : nat \\
  x : nat &\mid λx:nat.(+ (1 + x)) : nat → nat
\end{align*}\]

What if we try to type \(λx:nat.xx\)?

\[\begin{align*}
  x : nat &\mid ∈ \{ x : nat\} \\
  x : nat &\mid x : nat → τ \\
  x : nat &\mid x : nat \\
  x : nat &\mid xx : τ
\end{align*}\]

It should be reasonably obvious that we can quickly well-type (or prove untypable) any simply typed term. (Did the stage when σ turned into nat remind you of anything?)
Properties of simple types

▶ **Uniqueness of types.** In a given context (types for free variables), any simply typed lambda term has at most one type (and this is decidable, in small polynomial time). (This should be fairly obvious from what we’ve just done; formal proof takes a little work, but not much.)

▶ **Type safety.** The type of a term remains unchanged under $\alpha, \beta, \eta$-conversion. (Also straightforward, though with some lemmas (e.g. to handle substitution.).)

▶ **Strong normalization.** A well typed term evaluates under $\beta$-reduction in finitely many steps to a unique irreducible term. If the type is a base type, then the irreducible term is a constant. (A bit harder to prove.)

▶ **Corollary:** Simply typed lambda calculus cannot be Turing-complete! What have we lost?

---

Recursion lost

Just as $\lambda x.xx$ can’t be typed, nor can $Y$, nor any other potentially non-terminating term. (**Exercise:** convince yourself that $Y$ can’t be given a simple type.)

What can we do? General recursion is bound to allow non-termination.

---

Recursion re-gained

If your tool-kit doesn’t have a gadget: make one and put it in the kit!

▶ We add the term constructor $fix$ to the language: if $t$ is a term, then $(fix\ t)$ is a term.

▶ We extend $\beta$-reduction to say that:

$$fix(\lambda \cdot \cdot.t) \xrightarrow{\beta} t[fix(\lambda \cdot \cdot.t)/\cdot]$$

(often called the unfolding rule).

▶ We add a new typing rule:

$$\Gamma \vdash \cdot \cdot : \tau \rightarrow \tau$$

$$\Gamma \vdash fix\ t : \tau$$

Now we can use $fix$ just as we used $Y$.

**Exercise:** Define addition of (base type nat) numbers using $fix$, the successor and predecessor functions $suc, prd : nat \rightarrow nat$, and the equality of numbers function $= : nat \rightarrow nat \rightarrow nat$. 

---

More bells and whistles

We can further extend the syntax of simply typed lambda with constructors and types for the familiar apparatus of programming languages:

▶ tuples (products)
▶ records (named products)
▶ sums (union types)
▶ lists, trees etc. (N.B. one list type for each base type!)

See Pierce ch. 11 for gory (rather uninteresting) details.

More interesting is the let statement. Simple (non-recursive) lets can be done by:

$$\text{let } x : \tau = t \text{ in } t' \equiv (\lambda x : \tau.t)t$$

and fully recursive let definitions by:

$$\text{letrec } x : \tau = t \text{ in } t' \equiv \text{let } x : \tau = fix(\lambda x : \tau.t) \text{ in } t'$$

**Exercise:** Think carefully about the last! How does it work?
Type reconstruction

It’s rather boring having to type all these types! Can we leave them out, write in the untyped language, and let the computer work out the types? Essentially we program up the inference we did ‘in our heads’ to make type proofs.

Give every (ordinary) variable x a different type variable, so x:α. Then see what relations have to hold between the α.

Before, we were using meta-variables σ, τ to do human reasoning; now we’re going to promote these to actual variables ranging over types.

This is a constraint solving problem: e.g. we have α, β and we know that α = β → β.

Where do the constraints come from? From the typing rules.

First, some formalities . . .

Type variables

We add type variables to our typed language. Convention: I use the start of the Greek alphabet for type variables. Now types include:

- every type variable α is a type.

Eventually, type variables need to turn into actual types: consider (λx:α.x)1.

A type substitution gives a substitution of types for type variables, e.g. [nat/α, bool/β, (α → β)/γ]. We’ll write type substitution before a term, e.g. [nat/α](λx:α.x) = λx:nat.x

Substitutions apply to terms, contexts, etc. in the obvious way.

- Theorem: type substitution preserves typing: if Γ ⊢ t : τ, and θ is a type substitution, then Γ[θ] ⊢ T[θ] : Tτ
- The reverse is not true: x : α ⊬ x : nat, but [nat/α](x : α) ⊬ [nat/α]x : [nat/α]nat.

Reconstruction example

Find types for λf.λx.λy.suc(f (+ x y)).

First annotate: λf : γ.λx : α.λy : β.suc(f (+ x y)).

Now make the proof tree, using additional type variables and recording equations instead of working things out:

[α → θ → η = nat → nat → nat]
... ⊢ + : nat → nat → nat
... ⊢ x : α
... ⊢ y : β
[γ = η → nat]... ⊢ f : γ... ⊢ (+ x) : θ → η
... ⊢ (+ x y) : η
... ⊢ suc : nat → nat
... ⊢ f (+ x y) : nat [ζ = nat]

Solve the equations to get α = β = nat, and γ = nat → nat.

Solving the type equations

The equations are very simple: just equalities involving variables and →.

We can solve them by unification (technique from logic programming). Standard algorithm: look it up in Pierce for this case, or anywhere for the general case.

Theorem: if an untyped λ-term can be simply typed, this procedure will find a simple type for it, or return ‘fail’ otherwise (if the equations have no solution).

Proof by a somewhat technically involved induction on terms. (Full details in Pierce.)

So type inference is decidable (with reasonable polynomial complexity – in all practical cases, it’s even linear).
General solutions

- Consider the identity function \( \lambda x.x \). Type reconstruction gives \( \lambda x: \alpha . x : \alpha \) with no equations!
- What does an unsolved variable type mean? We can type \( \lambda x.x \) with any substitution \([\tau/\alpha]\).
- In general, the reconstruction algorithm gives us the most general type, leaving free variables.
- What about \( \lambda f : \alpha . \lambda x : \beta . f (fx) \)?
  - We get \( \alpha = \beta \rightarrow \beta \), so type \( (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta \).
- What about \( \lambda f : \alpha . \lambda x : \beta . f (x) \)?
  - We get \( \alpha = \beta \rightarrow \gamma \), so type \( (\beta \rightarrow \gamma) \rightarrow \beta \rightarrow \gamma \).
- But still, to get a real simply typed term, we must substitute.

Let-polymorphism

Allowing type schemes seems natural, but . . .

- could we still do type inference?
- are we going to get more complicated types?

We’ll come back to this, but just think about trying to type \( \lambda x : \text{nat}. xx \) – did anything happen in there that looked like type substitution?

Polymorphism

In the simply-typed \( \lambda \)-calculus, we have infinitely many distinct identity functions: \( \lambda x : \text{nat}. x \), \( \lambda x : \text{bool}. x \), \( \lambda x : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}. x \) and so on.

Wouldn’t be nice if we could write a polymorphic \( \lambda x.x \) that was actually the identity for all the types at once?

How did we avoid giving a type to the recursion ‘combinator’ \( \text{fix} \)?

Note that if we write it out each time, we can do this, as type reconstruction works separately on each instance:

\[
(\lambda x.x)(1), (\lambda x.x)(\text{true})
\]

What we can’t do is:

\[
\text{let id} = \lambda x.x \text{ in } (\text{id}(1), \text{id}(\text{true}))
\]

Why not? Un-sugar the \text{let} into a \( \lambda \)-application, and work through the reconstruction. How does it fail?

Polymorphism more formally

We have several possibilities for types with free type variables:

- some terms, like \( \lambda x : \alpha . x \) are well-typed ‘with the variables themselves’, and so are well-typed for any type substitution;
- some terms, like \( \lambda x : \alpha . (+ 1 x) \) are ill-typed, and only become well-typed for a specific substitution \([\text{nat}/\alpha]\);
- and some become well-typed for many substitutions:

\[
\lambda f : \alpha . \lambda x : \beta . f (fx)
\]

well types under \( T \) whenever \( T(\alpha) = T(\beta) \rightarrow T(\beta) \).

So what is the best type for \( \lambda x.x \)? Our typing rules tell us that for any \( \alpha \), we have \( \vdash (\lambda x : \alpha . x) : \alpha \rightarrow \alpha \). Perhaps we could say that \( \lambda x.x \) has the type scheme \( \forall \alpha. \alpha \rightarrow \alpha \)?

\( \lambda x . (+ 1 x) \), however, must have type \( \text{nat} \rightarrow \text{nat} \).

And \( \lambda f . \lambda x.f (fx) \) could have the type (scheme) \( \forall \beta . (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta \)?

The universal quantifier \text{binds} the free type variables.

Now go back to our original simply typed proof system, when we tried to type \( \lambda x : \text{nat}. xx \) – did anything happen in there that looked like type substitution?
Hindley–Milner type inference

First, we add let as a primitive:

- If \( x \) is a variable, and \( t \) and \( t' \) are terms, then \( \text{let } x = t \text{ in } t' \) is a term.
- It evaluates just as \( \lambda \):

\[
\text{let } x = t \text{ in } t' \Rightarrow t'[x/t]
\]

The difference comes in the typing rule:

\[
\Gamma \vdash t : \sigma \quad \Gamma, x : \tilde{\Gamma}(\sigma) \vdash t' : \tau
\]

\[
\Gamma \vdash \text{let } x = t \text{ in } t' : \tau
\]

What is \( \tilde{\Gamma}(\sigma) \)? It is the result of generalizing (binding with \( \forall \)) all free type variables in \( \sigma \) that do not occur free in \( \Gamma \).

E.g.: if \( x : \alpha \vdash t : \beta \), then \( \{x : \alpha\}(\beta) = \forall \beta. \alpha \rightarrow \beta \)

Generalization and Specialization

Generalization gave variables type schemes (or polytypes). For type inference to work, we have to un-generalize, or specialize, by modifying the rule for variables:

\[
\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \tau}
\]

What does \( \sigma \subseteq \tau \) mean? '

\( \tau \) is more special than \( \sigma \)', i.e. is got by substituting monotypes (no quantifiers!) for the quantified variables of \( \sigma \).

E.g. \( \forall \beta. \alpha \rightarrow \beta \subseteq \alpha \rightarrow \gamma \) and \( \forall \beta. \alpha \rightarrow \beta \subseteq \alpha \rightarrow (\text{nat} \rightarrow \text{bool}) \).

This lets every use of a polymorphic identifier get a new bunch of type variables to label it.

Polymorphic type inference example

Let's look at our motivating example

\[
\text{let } id = \lambda x. x \text{ in } \langle \text{id}(1), \text{id}(\text{true}) \rangle
\]

The inference tree is (with some abbreviation)

\[
\begin{array}{c}
\vdash \text{id}(1) : \alpha \\
\vdash \text{id}(\text{true}) : \delta \\
\vdash \text{let } id = \lambda x. x \text{ in } \langle \text{id}(1), \text{id}(\text{true}) \rangle : \alpha
\end{array}
\]

Remarks on Hindley–Milner

- The proof-tree plus unification description is convenient for proving properties of the system. Implementations do more or less the same (as the quite efficient version we've given).
- Type inference for let-polymorphism runs quickly (linearly) on every program a real programmer has ever written.
- But in fact it is \( \text{ExpTime} \)-complete!
- We can imagine generalizing to full polymorphism: allow all variables and terms to have actual types like \( \forall \alpha. \alpha \rightarrow \alpha \). Then \( (\lambda f.(f(1), f(\text{true}))(\text{nat} \rightarrow \text{bool}))(\lambda x.x) \) is fine.
- But type inference becomes undecidable!
- So we write the types explicitly, as in Java generics, and introduce type abstraction and type application:

\[
\begin{array}{c}
\frac{t : \tau}{(\\Lambda \alpha.t)(\alpha)} \Rightarrow [\tau/\alpha]t \\
\frac{(f(\text{bool}))(\text{true}),(f(\text{nat}))(1))((\Lambda \alpha.\lambda x : \alpha.x)}{\text{System F} \text{ (Girard), or polymorphic } \lambda\text{-calculus} \text{ (Reynolds).}}
\end{array}
\]