COMP20121 The Implementation and Power of Computer Languages

‘Power’ Part

http://www.cs.man.ac.uk/~petera/2121/index.html .

Peter Aczel

room: CS2.52, tel: 56155

email: petera@cs.man.ac.uk

School of Computer Science, University of Manchester

COMP20121, ‘power’ part: section 4

LECTURE TEN

Section 4: Computability ctd

The previous lecture

In the previous lecture we started looking at TMs as computational devices.

The previous lecture

In the previous lecture we started looking at TMs as computational devices. We considered

- Variants of TMs,
- The power of TMs,
- Simulating a von Neumann machine.

Plan of this lecture

In this lecture we consider

- Models of computation
  - The Church-Turing Thesis
- The power of computers
  - The Halting Problem
- Unrecognisable languages
- Recognisable, but undecidable languages

Models of computation

T here are many more models of computation than Turing machines and von Neumann machines.

Models of computation

T here are many more models of computation than Turing machines and von Neumann machines. See Exercises 30 and 31 for other examples.

Models of computation

T here are many more models of computation than Turing machines and von Neumann machines. See Exercises 30 and 31 for other examples.

And, of course, there are many computer languages we might use in order to compute something, such as C and Java.

Models of computation

An obvious question then is whether some forms of computations are more powerful than others.

Models of computation

An obvious question then is whether some forms of computations are more powerful than others.

The Church-Turing Thesis says that most of these notions lead to the same result:

For example, TMs are precisely as powerful as C or Java programs.

The power of computers

Are there problems which no computer can solve?

The power of computers

Are there problems which no computer can solve?

In C or Java a program may run forever (well, until it runs out of memory), and we have already seen that Turing machines may not halt.

The power of computers

It would be nice to have a compiler that would be able to detect this at compile-time, rather than us having to wait until run-time. Even at run-time, we might not be sure whether the program just took too long, or whether there is a non-terminating loop somewhere.

The power of computers

The question of deciding whether a given program (or TM) will terminate for a given input is known as the Halting Problem.

The Halting Problem

Assume we have written a program `Halt` which takes two files as its input.

The Halting Problem

Assume we have written a program `Halt` which takes two files as its input.

- In `file1`, it expects the code for a program.

The Halting Problem

Assume we have written a program `Halt` which takes two files as its input.

- In `file1`, it expects the code for a program.
- In `file2`, it expects the input for the program held in `file1`.

The Halting Problem

Hence a call of `Halt` has the form

```
Halt file1 file2.
```

The Halting Problem

Hence a call of `Halt` has the form

```
Halt file1 file2.
```

What is the behaviour of this program? We assume that when we run

```
Halt file1 file2
```

the program will always terminate. There are two possible outcomes.

The Halting Problem

- The program prints ‘does terminate’, which means that if we run the program in `file1` on the input in `file2`, that program will terminate.

The Halting Problem

- The program prints ‘does terminate’, which means that if we run the program in `file1` on the input in `file2`, that program will terminate.

- The program prints ‘does not terminate’, so that if we run the program in `file1` on the input in `file2`, that program will not terminate.

A new program, DHalt

We now change the program Halt and thus create a new program which we call DHalt. We do this by taking the code for Halt and making the following two kinds of modifications:

A new program, `DHalt`

- Whenever we find a print statement of ‘does terminate’ we replace it by a non-terminating loop

A new program, DHalt

- Whenever we find a print statement of ‘does terminate’ we replace it by a non-terminating loop, for example

```
x=0;while (x>=0) do x=x+1;
```

A new program, `DHalt`

- Whenever we find a print statement of ‘does terminate’ we replace it by a non-terminating loop .
- Whenever we find a print statement of ‘does not terminate’ we remove the ‘not’, so that instead ‘does terminate’ will be printed.

A new program, `DHalt`

Note that these are purely syntactic operations which merely involve going through the code looking for print instructions. Also note that the only way for `DHalt` not to terminate is by running into one of the loops we have put into the code of `Halt` to create it.

A new program, `DHalt`

We make one more change: We assume that `DHalt` only takes one input file `file`, which it uses in the place of both, `file1` and `file2`. Hence a call of `DHalt` looks like this:

```
DHalt file.
```

The behaviour of `DHalt`

So how can the new program `DHalt` behave? Again there are two cases.

The behaviour of `DHalt`

So how can the new program `DHalt` behave? Again there are two cases.

- Either the program runs forever. (In that case, the program in the input file will terminate when run on the input in the input file.)

The behaviour of `DHalt`

- Or the program will eventually print ‘does terminate’. (In that case, the program in the input file will not terminate when run on the input in the input file.)

The behaviour of `DHalt`

Now assume we have a file `dhalt` which contains a copy of the code for `DHalt`.

We try to find out what happens if we run `DHalt dhalt`. Again there are two cases.

Self-application

- The program terminates. Then

Self-application

- The program terminates. Then
  - `DHalt` prints ‘does terminate’.

Self-application

- The program terminates. Then
  - `DHalt` prints ‘does terminate’.
  - Hence `Halt dhalt dhalt` prints ‘does not terminate’.

Self-application

- The program terminates. Then
  - `DHalt` prints ‘does terminate’.
  - Hence `Halt dhalt dhalt` prints ‘does not terminate’.
  - That means that the program in `dhalt` (which is `DHalt`) will not terminate when run on the input `dhalt`.

Self-application

- The program terminates. Then
  - `DHalt` prints ‘does terminate’.
  - Hence `Halt dhalt dhalt` prints ‘does not terminate’.
  - That means that the program in `dhalt` (which is `DHalt`) will not terminate when run on the input `dhalt`.

This is a contradiction.

Self-application

- The program does not terminate. Then

Self-application

- The program does not terminate. Then
  - `Halt dhalt dhalt` prints ‘does terminate’.

Self-application

- The program does not terminate. Then
  - `Halt dhalt dhalt` prints ‘does terminate’.
  - That means that the program in `dhalt` (which is `DHalt`) will terminate when run on the input `dhalt`.

Self-application

- The program does not terminate. Then
  - `Halt dhalt dhalt` prints ‘does terminate’.
  - That means that the program in `dhalt` (which is `DHalt`) will terminate when run on the input `dhalt`.

This is a contradiction.

Self-application

- The program does not terminate. Then
  - `Halt dhalt dhalt` prints ‘does terminate’.
  - That means that the program in `dhalt` (which is `DHalt`) will terminate when run on the input `dhalt`.

  This is a contradiction.

So a program like `Halt` cannot exist!

Encoding Turing machines

If we want to apply a similar trick to Turing machines we have to be able to use a TM as the input to another TM.

Encoding Turing machines

If we want to apply a similar trick to Turing machines we have to be able to use a TM as the input to another TM. In other words, we have to find a syntactic description for Turing machines. This is also known as an encoding. For this we have to decide what the underlying alphabet should be. This could, for example, be a binary encoding, that is over the alphabet $\{0, 1\}$.

Encoding Turing machines

We need to be able to encode the transition table, that is we need to be able to encode

Encoding Turing machines

We need to be able to encode the transition table, that is we need to be able to encode

- States:

Encoding Turing machines

We need to be able to encode the transition table, that is we need to be able to encode

- States: Whichever underlying alphabet we choose, this shouldn't be hard.

Encoding Turing machines

We need to be able to encode the transition table, that is we need to be able to encode

- States: Whichever underlying alphabet we choose, this shouldn't be hard.
- Tape symbols:

Encoding Turing machines

We need to be able to encode the transition table, that is we need to be able to encode

- States: Whichever underlying alphabet we choose, this shouldn't be hard.
- Tape symbols: We need to be able to encode these in the underlying alphabet.

Encoding Turing machines

- Movement symbols L, R and N: No difficulty here.

Encoding Turing machines

- Movement symbols L, R and N: No difficulty here.
- Starting state, possibly accepting states.

Encoding Turing machines

- Movement symbols L, R and N: No difficulty here.
- Starting state, possibly accepting states.

Every entry in the transition table can be described by a quintuple

$\langle$no of state$\rangle$ $\langle$current symbol$\rangle$ $\langle$no of new state$\rangle$ $\langle$symbol to write$\rangle$ $\langle$move$\rangle$.

Encoding Turing machines

We may assume that there is an extra symbol, *e.g.* $\#$, to separate entries from each other. We can list start and accepting states before or after the quintuples. Then a TM can be encoded as one long string over the chosen alphabet with $\#$ added.

Encoding Turing machines

We use $\langle M \rangle$ to refer to the encoding of a machine M. It is customary to choose an encoding within the tape alphabet for M.

How many strings over Σ?

The fact that we can encode Turing machines into strings over some alphabet tells us something about them: There can only be countably many of them.

How many strings over Σ?

The fact that we can encode Turing machines into strings over some alphabet tells us something about them: There can only be countably many of them.

Let Σ be any alphabet. Then there are infinitely many strings over this alphabet (even if it consists of just one letter!), but this is a countable infinity.

How many strings over Σ?

Countably infinite means that we can list these strings, which means that we can number them: There's a first, a second, a third, and so on.

How many strings over Σ?

Countably infinite means that we can list these strings, which means that we can number them: There's a first, a second, a third, and so on.

This numbering proceeds in such a way that when you give me any string you choose, I must be able to tell you which number it will get.

How many strings over Σ?

Countably infinite means that we can list these strings, which means that we can number them: There's a first, a second, a third, and so on.

This numbering proceeds in such a way that when you give me any string you choose, I must be able to tell you which number it will get. In other words, our list of strings contains every string there is.

How many strings over Σ?

This listing works as follows: First give all the strings consisting of one symbol, then all the strings consisting of two symbols, then all the strings consisting of three symbols,. . . .

How many strings over Σ?

This listing works as follows: First give all the strings consisting of one symbol, then all the strings consisting of two symbols, then all the strings consisting of three symbols,. . . .

If Σ has n elements, then there are n different strings consisting of one letter, n^2 consisting of two letters, n^3 many consisting of three letters, and so on.

How many strings over Σ?

This demonstrates that the set Σ^* of all strings over Σ is countably infinite.

How many Turing machines?

- Clearly not every string over our underlying alphabet will be a proper encoding of a Turing machine. Many of them will be meaningless if we try to interpret them as Turing machines.

How many Turing machines?

- Clearly not every string over our underlying alphabet will be a proper encoding of a Turing machine. Many of them will be meaningless if we try to interpret them as Turing machines.
- Nonetheless we can use the list of all strings to obtain a list of all Turing machines: We just leave out the strings which don't encode a machine, and adjust the numbers accordingly.

How many Turing machines?

It should be clear that there are infinitely many TMs over any given alphabet, and the above argument tells us that we are once again talking about a countable infinity.

How many languages over Σ?

- A language over Σ is a subset of Σ^*,

which we know to be countably infinite.

How many languages over Σ?

- A language over Σ is a subset of Σ^*, which we know to be countably infinite.
- Hence we are asking: How many subsets does a countably infinite set have?

How many languages over Σ?

- A language over Σ is a subset of Σ^*, which we know to be countably infinite.
- Hence we are asking: How many subsets does a countably infinite set have?
- Clearly there must be infinitely many: Merely the one-element subsets provide us with infinitely many subsets.

How many languages over Σ?

- A language over Σ is a subset of Σ^*, which we know to be countably infinite.
- Hence we are asking: How many subsets does a countably infinite set have?
- Clearly there must be infinitely many: Merely the one-element subsets provide us with infinitely many subsets.
- But this time, we have an uncountable infinity.

How many subsets of a countably. . .

Let A be an infinite countable set, with elements a_1, a_2, . . . Because we know that A is infinite, we know that this list goes on forever.

How many subsets of a countably. . .

Let A be an infinite countable set, with elements a_1, a_2, . . . Because we know that A is infinite, we know that this list goes on forever.

Assume that we have a list which contains all the subsets of A, starting with A_1, A_2, . . .

How many subsets of a countably. . .

Let A be an infinite countable set, with elements a_1, a_2, . . . Because we know that A is infinite, we know that this list goes on forever.

Assume that we have a list which contains all the subsets of A, starting with A_1, A_2, . . . We can describe every such set by saying which of the elements of A belong to it.

How many subsets of a countably. . .

For A_1, this might look as follows.

Set\element	a_1	a_2	a_3	a_4	a_5	a_6	. . .
A_1		✓		✓	✓		. . .

How many subsets of a countably. . .

We can describe our entire list of sets in this way.

Set\element	a_1	a_2	a_3	a_4	a_5	a_6	. . .
A_1		✓		✓	✓		. . .
A_2	✓	✓					. . .
A_3		✓	✓	✓	✓		. . .
A_4			✓		✓		. . .
⋮							

How many subsets of a countably. . .

We describe a set B by stipulating its members.

Set\element	a_1	a_2	a_3	a_4	a_5	a_6	. . .
A_1		✓		✓	✓		. . .
A_2	✓	✓					. . .
A_3		✓	✓	✓	✓		. . .
A_4			✓		✓		. . .
⋮							
B	?	?	?	?	?	?	. . .

How many subsets of a countably. . .

Because a_1 is not in A_1, we decide that it should be in B.

Set\element	a_1	a_2	a_3	a_4	a_5	a_6	...
A_1		✓		✓	✓		...
A_2	✓	✓					...
A_3		✓	✓	✓	✓		...
A_4			✓		✓		...
⋮							
B	✓	?	?	?	?	?	...

How many subsets of a countably. . .

Because a_2 is in A_2, we decide that it should not be in B.

Set\element	a_1	a_2	a_3	a_4	a_5	a_6	...
A_1		✓		✓	✓		...
A_2	✓	✓					...
A_3		✓	✓	✓	✓		...
A_4			✓		✓		...
⋮							
B	✓		?	?	?	?	...

How many subsets of a countably. . .

Because a_3 is in A_3, we decide that it should not be in B.

Set\element	a_1	a_2	a_3	a_4	a_5	a_6	...
A_1		✓		✓	✓		...
A_2	✓	✓					...
A_3		✓	✓	✓	✓		...
A_4			✓		✓		...
⋮							
B	✓			?	?	?	...

How many subsets of a countably. . .

Because a_4 is not in A_4, we decide that it should be in B.

Set\element	a_1	a_2	a_3	a_4	a_5	a_6	...
A_1		✓		✓	✓		...
A_2	✓	✓					...
A_3		✓	✓	✓	✓		...
A_4			✓		✓		...
⋮							
B	✓			✓	?	?	...

How many subsets of a countably. . .

It should be clear now how to continue with the definition of B:

a_n is in B iff a_n is not in A_n.

B does not appear on the list

When we compare B with A_1, we note that they differ as far as element a_1 is concerned.

Set\element	a_1	a_2	a_3	a_4	a_5	a_6	...
A_1		✓		✓	✓		...
A_2	✓	✓					...
A_3		✓	✓	✓	✓		...
A_4			✓		✓		...
⋮							
B	✓			✓	?	?	...

B does not appear on the list

When we compare B with A_2, we note that they differ as far as element a_2 is concerned.

Set \ element	a_1	a_2	a_3	a_4	a_5	a_6	...
A_1		✓		✓	✓		...
A_2	✓	✓					...
A_3		✓	✓	✓	✓		...
A_4			✓		✓		...
⋮							
B	✓			✓	?	?	...

B does not appear on the list

When we compare B with the nth element A_n, we note that they differ as far as element a_n is concerned.

Set\element	a_1	a_2	a_3	a_4	a_5	a_6	...
A_1		✓		✓	✓		...
A_2	✓	✓					...
A_3		✓	✓	✓	✓		...
A_4			✓		✓		...
⋮							
B	✓			✓	?	?	...

B does not appear on the list

Hence B does not appear on the list!

B does not appear on the list

Hence B does not appear on the list!

This means that our list never contained all the subsets of A. Since we assumed it was an arbitrary such list, we have to deduce that such a list cannot exist.

So A has uncountably many subsets.

B does not appear on the list

This method for showing that a list does not contain everything it was supposed to contain is known as the diagonalization method.

Unrecognizable languages

We have now demonstrated the following:

Unrecognizable languages

We have now demonstrated the following:

- There are countably infinitely many Turing machines. Hence there are at most countably infinitely many Turing-recognizable languages (the languages recognized by these TMs).

Unrecognizable languages

We have now demonstrated the following:

- There are countably infinitely many Turing machines. Hence there are at most countably infinitely many Turing-recognizable languages (the languages recognized by these TMs).
- There are uncountably many languages over a given alphabet.

Unrecognizable languages

Hence there must be uncountably many languages which are not Turing-recognizable!

An undecidable language

If we translate the idea of the program `DHalt` for Turing machines, the question could be changed to:

An undecidable language

If we translate the idea of the program `DHalt` for Turing machines, the question could be changed to:

Can we have a Turing machine which, given the code $\langle M \rangle$ for a TM M and an input string α, will decide whether or not M accepts α?

An undecidable language

This is a variation of the Halting Problem, just for Turing machines. We could alternatively choose a TM which, given the code $\langle M \rangle$ for a TM M and an input string α, will decide whether or not M halts for input α. (Note that in order to accept α, M will have to halt.) We will look at this question later.

An undecidable language

Theorem 4.1 *The language*

$$L_{TM} = \{(\langle M \rangle, \alpha) \mid M \text{ is a TM which accepts } \alpha\}$$

is not decidable.

An undecidable language

Theorem 4.1 *The language*

$$L_{TM} = \{(\langle M \rangle, \alpha) \mid M \text{ is a TM which accepts } \alpha\}$$

is not decidable.

You should have expected this result, because of the Church-Turing thesis and its similarity to the Halting Problem.

A recognizable but undecidable lang.

The language

$$L_{\text{TM}} = \{(\langle M \rangle, \alpha) \mid M \text{ is a TM which accepts } \alpha\}$$

is not decidable, but it is

Turing-recognizable!

A recognizable but undecidable lang.

The language

$$L_{\text{TM}} = \{(\langle M \rangle, \alpha) \mid M \text{ is a TM which accepts } \alpha\}$$

is not decidable, but it is

Turing-recognizable!

We will only give a brief argument for this.

A recognizable but undecidable lang.

There is a ‘universal Turing machine’ U

which can simulate every other TM!

A recognizable but undecidable lang.

There is a 'universal Turing machine' U which can simulate every other TM! Given an encoding $\langle M \rangle$ for a TM M and a string α, U will behave like M for the input string α. U recognizes the language L_{TM}.

A recognizable but undecidable lang.

To see how U works, assume that it has one tape with the encoding $\langle M \rangle$ of M, and another tape with the input string α. What it then does is to use the first tape to read off what to do. It carries out those actions on the second tape, thus mimicking M's action.

A recognizable but undecidable lang.

If U thus works out that M accepts α (which requires M to halt) it will accept the pair $(\langle M \rangle, \alpha)$.

A recognizable but undecidable lang.

If U thus works out that M accepts α (which requires M to halt) it will accept the pair $(\langle M \rangle, \alpha)$.

If M goes on forever, so will U!

Note that U recognizes L_{TM}, but does not decide it.