Undecidability

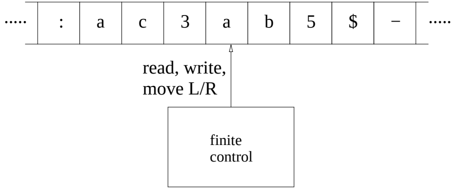
Informatics 2A: Lecture 31

Mary Cryan

School of Informatics
University of Edinburgh
mcryan@inf.ed.ac.uk

28 November 2018

Recap: Turing machines



- If $|\Sigma| \geq 2$, any kind of 'finite data' can be coded up as a string in Σ^*, which can then be written onto a Turing machine tape. (E.g. natural numbers could be written in binary.)
- According to the Church-Turing thesis (CTT), any 'mechanical computation' that can be performed on finite data can be performed in principle by a Turing machine.
- Any decent programming language (and even Micro-Haskell!) has the same computational power in principle as a Turing machine.

Universal Turing machines

Consider any Turing machine with input alphabet Σ.

Such a machine T is itself specified by a finite amount of information, so can in principle be 'coded up' by a string $\overline{T} \in \Sigma^*$. (Details don't matter).

So one can imagine a universal Turing machine U which:

- Takes as its input a coded description $\overline{T}$ of some TM T, along with an input string s, separated by a blank symbol.
- Simulates the behaviour of T on the input string s. (N.B. a single step of T may require many steps of U.)
  - If T ever halts (i.e. enters final state), U will halt.
  - If T runs forever, U will run forever.

If we believe CTT, such a U must exist — but in any case, it's possible to construct one explicitly.

The concept of a general-purpose computer

Alan Turing's discovery of the existence of a universal Turing machine (1936) was in some sense the fundamental insight that gave us the general-purpose (programmable) computer.

In most areas of life, we have different machines for different jobs. So it's quite remarkable that a single physical machine can be persuaded to perform as many different tasks as a computer can . . . just by feeding it with a cunning sequence of 0's and 1's!

The halting problem

The universal machine U in effect serves as a recognizer for the set

$$\{\overline{T}\_s \mid T \text{ halts on input } s\}$$

But is there also a machine V that recognizes the set

$$\{\overline{T}\_s \mid T \text{ doesn't halt on input } s\} \text{ ?}$$

If there were, then given any T and s, we could run U and V in parallel, and we'd eventually get an answer to the question "does T halt on input s?"

Conversely, if there were a machine that answered this question, we could construct a machine V with the above property.

Theorem: There is no such Turing machine V!
In other words, the halting problem is undecidable.

Proof of undecidability

Why is the halting problem undecidable?

Suppose V existed. Then we could easily make a Turing machine W that recognised the set L defined by:

$$L = \{s \in \Sigma^* \mid \text{ the TM coded by } s \text{ runs forever on the input } s\}$$

(W could construct the string s_s, then run as V on it.)

Now consider what W does when given the string $\overline{W}$ as input. That is, the input to W is the string that encodes W itself.

- W accepts $\overline{W}$ iff W runs forever on $\overline{W}$ (since W recognises L)
- but W accepts $\overline{W}$ iff W halts on $\overline{W}$ (definition of acceptance)

Contradiction!!! So V can't exist after all!

Precursor: Russell's paradox (1901)

Define R to be the set of all sets that don't contain themselves:

$$R = \{S \mid S \notin S\}$$

Does R contain itself, i.e. is $R \in R$?

Russell's analogy: The village barber shaves exactly those men in the village who don't shave themselves. Does the barber shave himself, or not?

Precursor: Russell's paradox (1901)

Define R to be the set of all sets that don't contain themselves:

$$R = \{S \mid S \notin S\}$$

Does R contain itself, i.e. is $R \in R$?

Conclusion: no such set R exists.

Russell's analogy: The village barber shaves exactly those men in the village who don't shave themselves. Does the barber shave himself, or not?

Precursor: Russell's paradox (1901)

Define R to be the set of all sets that don't contain themselves:

$$R \ = \ \{S \mid S \notin S\}$$

Does R contain itself, i.e. is $R \in R$?

Conclusion: no such set R exists.

Russell's analogy: The village barber shaves exactly those men in the village who don't shave themselves. Does the barber shave himself, or not?

Conclusion: no man exists in the village with the property proposed by Russell.

Precursor: Russell's paradox (1901)

Define R to be the set of all sets that don't contain themselves:

$$R = \{S \mid S \notin S\}$$

Does R contain itself, i.e. is $R \in R$?

Conclusion: no such set R exists.

Russell's analogy: The village barber shaves exactly those men in the village who don't shave themselves. Does the barber shave himself, or not?

Conclusion: no man exists in the village with the property proposed by Russell.

Highly recommended reading: *Scooping the Loop Snooper* by Geoffrey Pullum. (A proof that the Halting Problem is undecidable, written in verse in the style of Dr. Seuss).

Decidable vs. semidecidable sets

In general, a set S (e.g. $\subseteq \Sigma^*$) is called decidable if there's a mechanical procedure which, given $s \in \Sigma^*$, will always return a yes/no answer to the question "Is $s \in S$?".
E.g. the set $\{s \mid s \text{ represents a prime number}\}$ is decidable.

We say S is semidecidable if there's a mechanical procedure which will return 'yes' precisely when $s \in S$ (it isn't obliged to return anything if $s \notin S$).

Semidecidable sets coincide with recursively enumerable (=Type 0) languages as defined in lectures 28–9.

The halting set $\{\overline{T}\text{␣}s \mid T \text{ halts on input } s\}$ is an example a semidecidable set that isn't decidable. So there exist Type 0 languages for which membership is undecidable.

Separating Type 0 and Type 1

Every Type 1 (context-sensitive) language is decidable.
(The argument was outlined in Lecture 29.)

As we have seen, the halting set

$$\{\overline{T}\,\_\,s \mid T \text{ halts on input } s\}$$

is an undecidable Type 0 language.

So the halting set is an example of a Type 0 language that is not a Type 1 language.

(Last lecture, we saw another example: the set of provable sentences of FOPL. This too is an undecidable Type 0 language.)

Undecidable problems in mathematics

The existence of ‘mechanically unsolvable’ mathematical problems was in itself a major breakthrough in mathematical logic: until about 1930, some people (the mathematician David Hilbert in particular) hoped there might be a single killer algorithm that could solve all mathematical problems!

Once we have one example of an unsolvable problem (the halting problem), we can use it to obtain others — typically by showing “the halting problem can be reduced to problem X.”
(If we had a mechanical procedure for solving X, we could use it to solve the halting problem.)

Example: Provability of theorems

Let M be some reasonable (consistent) formal logical system for proving mathematical theorems (something like Peano arithmetic or Zermelo-Fraenkel set theory).

Theorem: The set of theorems provable in M is semidecidable (and hence is a Type 0 language), but not decidable.

Proof: Any reasonable system M will be able to prove all true statements of the form "T halts on input s". So if we could decide M-provability, we could solve the halting problem.

Corollary (Gödel): However strong M is, there are mathematical statements P such that neither P nor $\neg P$ is provable in M.

Proof: Otherwise, given any P we could search through all possible M-proofs until either a proof of P or of $\neg P$ showed up. This would give us an algorithm for deciding M-provability.

Example: Diophantine equations

Suppose we're given a set of simultaneous equations involving polynomials in several variables with integer coefficients. E.g.

$$
\begin{aligned}
3xy + 4z + 5wx^2 &= 27 \\
x^2 + y^3 - 9z &= 4 \\
w^5 - z^4 &= 31 \\
x^2 + y^2 + z^2 - w^2 &= 2536427
\end{aligned}
$$

Hilbert's 10th Problem (1900): Is there a mechanical procedure for determining whether a set of polynomial equations has an integer solution?

Matiyasevich's Theorem (1970): It is undecidable whether a given set of polynomial equations has an integer solution.

(By contrast, it's decidable whether there's a solution in real numbers!)

Another example: Post correspondence problem

Given two finite sets S, T of strings, decide whether or not there's a string that can be formed both as a concatenation of strings in S and as a concatenation of strings in T.

E.g. suppose

$$S = \{a, ab, bba\}, \qquad T = \{baa, aa, bb\}$$

Then the answer is YES, because:

$$bba.ab.bba.a = bbaabbbaa = bb.aa.bb.baa$$

In general, however, it's undecidable whether such a string exists for a given S, T.

There are also examples from formal language theory itself.
E.g. given two context-free grammars G_1, G_2, it's undecidable whether $\mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ is context-free.

Bonus Topic: Higher-Order Computability

In one sense, all reasonable prog. langs are equally powerful. E.g.

- They can compute the same class of functions $\mathbb{Z} \rightharpoonup \mathbb{Z}$. (In Micro-Haskell, these have type `Integer->Integer`).
- Any language can be implemented in any other. (E.g. you've implemented MH in Java.)

Indeed, there's only one reasonable mathematical class of 'computable' functions $\mathbb{Z} \rightharpoonup \mathbb{Z}$ (the Turing-computable functions).

But what about higher-order functions, e.g. of type
`((Integer->Integer)->Integer)->Integer` ?

- What does it mean for a function of this kind to be 'computable'?
- Are all reasonable languages 'equally powerful' when it comes to higher-order functions?

Case study: iteration vs. recursion

Many tasks that involve 'looping' can be accomplished using either iteration or recursion. E.g. to compute the factorial function:

```
fac(n) {
   int m = 1 ;
   for i = 1 to n
      m = m * i ;
   return m
}
```

```
fac(n) {
   if n == 0
      return 1
   else
      return fac(n-1) * n
}
```

Just a matter of style? Or is there a deeper difference?

Consider the MH program:

```
G : (Integer -> Integer -> Integer) -> Integer -> Integer
G f n = f n (G f (n+1))
```

(Informally, `G f 0 = f 0 (f 1 (f 2 (...)))`.)

Theorem (Berger 1999, JL 2015). The 2nd order function `G` can't be computed by 'iteration alone': recursion is essential here.

Recursions at higher types

That definition again:

```
G : (Integer -> Integer -> Integer) -> Integer -> Integer
G f n = f n (G f (n+1))
```

The thing we're defining recursively here is really `G f`.

So for `G`, or indeed for `factorial`, we only need 'recursion at type `Integer->Integer`'. Is this all the recursion we *ever* need?

Let's write MH_k for the sublanguage of MH where we only allow recursions at types of order $\leq k$. So $MH_1 \subseteq MH_2 \subseteq \cdots \subseteq MH$.

All of these languages are Turing-complete, i.e. they yield the same computable functions of type `Integer->Integer`. But they differ in the higher-order functions that they can compute:

Theorem (JL 2015). For every k, there are higher-order functions computable in MH_{k+1} but not in MH_k.

That's it folks!

That concludes the course syllabus.

On Friday 28th, Shay and I will present a joint revision lecture, in which we shall discuss:

- the exam structure
- examinable material
- pointers to UG3 (and upwards) Informatics courses that continue from this one