Computability II - Universal Computation

CS605 - Math & Theory of CS
T.J. Naughton
Maynooth University, Ireland

Turing, defining computability

- The *computable numbers* (according to Turing) form a class defined as those infinite real numbers that can be printed by a TM starting with a blank tape.
- It can be proved that π and $\sqrt{2}$ are computable numbers, as well as every real number (irrational or otherwise) defined by the normal equations of mathematics.

© T J Naughton, Maynooth University, Ireland

Turing, defining computability

To summarise the previous slides:

- Each table of behaviour defines a new TM.
- A number is *computable* if one of the TMs computes it.
- A process/calculation is computable if it maps to one of the computable numbers under some global encoding scheme.

© T J Naughton, Maynooth University, Ireland

Turing, defining uncomputability

- Armed with this definition of computability, however, we can prove that there are some numbers which are not of the computable class; that some *uncomputable numbers* exist.

© T J Naughton, Maynooth University, Ireland

Turing, defining uncomputability

- Turing was the first person to define something that a computer could not do.
- He used Gödel's trick of self-reference.

© T J Naughton, Maynooth University, Ireland

Turing, defining uncomputability

- Every TM was treated as a machine that writes out a number rather than a machine that performs some calculations on numbers.
- [If two machines write out the same number, then we say that they are equivalent machines (just like two syntactically-different Java programs might be equivalent).]

© T J Naughton, Maynooth University, Ireland

Turing, defining uncomputability

- That number then represents that machine, and there are an infinite number of such machines, just as we might imagine.
- Turing then argues that the computable numbers form a countable set, as follows.

© T J Naughton, Maynooth University, Ireland

Turing, defining uncomputability

- The table of behaviour of each of Turing's machines must be *finite in length because*

State i	*Read*	*Write*	*Move*	*State i+1*

(i) no row in the look-up table has the same first two symbols, and *(ii)* both sets (states & symbols) are finite.

© T J Naughton, Maynooth University, Ireland

Turing, defining uncomputability

- Since the table of any of Turing's machines is finite in length, all possible tables of behaviour may be ordered (lexicographically, for example),

  and this shows that they are *countable* (or enumerable or countably infinite).

© T J Naughton, Maynooth University, Ireland

Note...

- Note: *countability* is a mathematical term independent of computer science or complexity theory. It relates to the cardinality of infinite sets. Be careful not to confuse it with *computability*.
- A set is *countable* if its elements can be mapped to a subset of **N** (a set is countable if it can be ordered).

  An infinite number is *computable* if a Turing machine can write out its value.

© T J Naughton, Maynooth University, Ireland

Turing, defining uncomputability

- Since the set of all numbers generated by TMs is countable, and the set of real numbers is obviously uncountable, it follows that *almost all* numbers are uncomputable.
- Turing therefore proves the existence of uncomputable numbers.

© T J Naughton, Maynooth University, Ireland

Turing, defining uncomputability

- This seems perfectly reasonable (if a little depressing for computer scientists).
- But now Turing outlines a sequence of steps to actually compute one of these uncomputable numbers.

© T J Naughton, Maynooth University, Ireland

Turing, defining uncomputability

- He defines the following 'algorithm:'
  1. Take the set of TMs and order them
  2. Remove all machines that only produce a finite number of digits (we are then left with all TMs that write out numbers with an infinite sequence of digits after the decimal place – the computable numbers).
  3. Use diagonalisation to construct a number not in this set – an uncomputable number.

© T J Naughton, Maynooth University, Ireland

Turing, defining uncomputability

- The surprising thing here is not that such a number exists (for almost all infinite-decimal-expansion numbers are not in the set) but that there is an algorithm to construct it.
- If this is an algorithm (e.g. a TM) to write out this particular number, why wasn't that particular number already in the set of infinite numbers that can be written out by TMs?

© T J Naughton, Maynooth University, Ireland

Turing, defining uncomputability

- There is a fundamental contradiction in the concept of computing an uncomputable number.
- There must be a flaw in Turing's 'algorithm.' It must not be an algorithm at all. At least one of its steps must be uncomputable.
- Now, the first step is surely computable. How could you do it?
- Also, the third step is computable. How could you do it?

© T J Naughton, Maynooth University, Ireland

Turing, the halting problem

- This means that the second step must be uncomputable.
- This is a proof that the second step is uncomputable.

© T J Naughton, Maynooth University, Ireland

Turing, the halting problem

- Turing cleverly arranged this flaw in his 'algorithm' so he could prove that *segregating those tables of behaviour which produce a finite sequence of digits* from those producing an infinite one *is uncomputable*.

  However simple it sounds, we must not be able to do it. That's the only possible cause of our contradiction!

© T J Naughton, Maynooth University, Ireland

Turing, the halting problem

- Identifying which tables of behaviour will produce an infinite sequence and which will not is *not* a computable operation
  - there is no TM which can inspect the table of behaviour of another machine and decide whether it will produce an infinite sequence of digits or not.
- Or more generally...

© T J Naughton, Maynooth University, Ireland

Turing, the halting problem

- If we construct a TM, which if given another machine's table of behaviour as part of its input can emulate that machine, then that TM can never definitively say whether the table of behaviour terminates/halts or not.
- Really? Well, the proof has just been shown. You have to accept it.

© T J Naughton, Maynooth University, Ireland

Turing, the halting problem

- If you want to reason about it, we could consider an obvious technique: one TM (call it **A**) emulating another (call it **B**) in order to decide if **B** terminates or not. **A**, however, must wait to make its decision on whether **B** terminates until after its emulation of **B** has terminated. So that technique has a flaw – it can tell if **B** terminates, but it can't tell if **B** does not terminate.
- There are other possible techniques, such as tricks to see if the condition of a while loop always remains true. However, you can appreciate it if you try to look for rules that you will need different tricks for different while loops (consider nested loops, etc).
- Turing's proof tells us that we'll actually need an infinite number of tricks to check the condition of all of the possible while loop set-ups. Algorithms can only be finite in length (a finite number of tricks), so no single algorithm could check all the loops.

© T J Naughton, Maynooth University, Ireland

Turing, the halting problem

- This was a very famous observation and, today, is called the *halting problem*
  – given a program, does it halt on all inputs?
- One can say that the halting problem cannot be decided by a TM (an additional formal proof follows soon).

© T J Naughton, Maynooth University, Ireland

Turing, the Entscheidungsproblem

- With the discovery of a problem that could not be decided by a machine

  <and a little mathematical logic>

  Hilbert's *Entscheidungsproblem* could finally be answered; computation (and mathematics) is *not decidable*.

© T J Naughton, Maynooth University, Ireland

Entscheidungsproblem, 1931

- Hilbert's third question had been
  – decidability; is there a mechanical method that can be applied to any mathematical assertion which will eventually tell whether that assertion is *provable* or not?

© T J Naughton, Maynooth University, Ireland

Turing, the Entscheidungsproblem

- So Turing defines his formal model of computation in order to answer Hilbert's final question.
- However, as Turing indicated by the title of his paper...

  "On computable numbers, with an application to the *Entscheidungsproblem*"

  ...answering Hilbert's question was just an application of his machines.

© T J Naughton, Maynooth University, Ireland

Turing, the Universal Machine

- We have already introduced the ability of TMs to inspect and assume another's table of behaviour.
- It is possible to construct such a TM, called a Universal Machine which may assume any TM's table.

© T J Naughton, Maynooth University, Ireland

Turing, the Universal Machine

- This allows all computable numbers to be computed by a single machine and this machine is therefore capable of executing any describable algorithm.

© T J Naughton, Maynooth University, Ireland

Universal machines

- When we speak of a Universal Machine we of course speak of the set of Universal Machines.
  - since an infinite number of different but functionally identical Universal Machines may be constructed.

© T J Naughton, Maynooth University, Ireland

Universal machines

- It is surprising how simply a universal computer can be defined.
- Minsky, in 1967, showed that 4 tape symbols and 7 states will suffice.

Q What can you say about the table of behaviour of Minsky's universal TM?

© T J Naughton, Maynooth University, Ireland

Universal machines

- As we can imagine, the number of tape symbols can be reduced still further, by increasing the number of states.

Q Where have we seen this in our own experience? What is the minimum number of tape symbols for computation?

- And we can in turn reduce this increased number of states by writing more of those symbols on the tape.

© T J Naughton, Maynooth University, Ireland

Point of interest (i)

- It has been said that "...most importantly, Turing's ideas differed from those of others who were solving arithmetic problems by introducing the concept of symbol processing."

Q Had we seen that concept previously?

© T J Naughton, Maynooth University, Ireland

Point of interest (ii)

Q What does this popular statement mean?

"If it is computable, then there is a Turing machine to compute it"

© T J Naughton, Maynooth University, Ireland

Point of interest (ii)

- It does not mean that if *we* regard something as "computable", we will eventually be able to construct a TM to compute it.
- As we have seen "computable" is a term with an unambiguous mathematical definition
  - Turing's machine actually *defines* what is computable.

© T J Naughton, Maynooth University, Ireland

Point of interest (ii)

- Those things which are "computable" already have a TM defined for them in the countable set of TMs.
- For clarity, the statement could be phrased

"If it is computable, then there already is a Turing machine to compute it"

© T J Naughton, Maynooth University, Ireland

Point of interest (ii)

- Constructing the correct TM could then be described in terms of a search through the countable set of TMs.
  - We cannot add something to this set without changing the technical meaning of computation.

© T J Naughton, Maynooth University, Ireland

Point of interest (ii)

Q If the set of TMs is already defined, should we be able to tell immediately if a given problem is computable or not?

Q If building the correct TM involves a search, how many correct TMs could we find in the set of TMs?

© T J Naughton, Maynooth University, Ireland

Point of interest (iii)

- There are other formal proofs of the undecidability of the Halting problem.

© T J Naughton, Maynooth University, Ireland

Halting problem

"Given a program *P* and an input *X*, does *P* halt on *X*?"

```
bool halts(string P, string X) {
  // if P halts on X return true
  // else return false
}

void newfn(string X) {
  while halts(X, X) {
  }
  printf("I'm halting now!");
}
```

© T J Naughton, Maynooth University, Ireland

Halting problem

- Does `newfn(newfn)` halt?
- Only if `halts(newfn, newfn)` returns false.
- So `newfn(newfn)` does halt only if `newfn(newfn)` does not halt?
- Our logic was flawless, therefore we must have started from an invalid premise: that `halts(P, X)` can exist.

© T J Naughton, Maynooth University, Ireland

Point of interest (iv)

- There are other simple constructions of universal TMs.

© T J Naughton, Maynooth University, Ireland

Turing, defining computability

- We pause now and detour from Turing's definition of computability.
- A definition of computability had already been presented by Church that year, and Turing had referred to it in his paper.

© T J Naughton, Maynooth University, Ireland

Alonzo Church, 1936

- The U.S. logician Alonzo Church extended the work of Gödel.
- He developed the λ-calculus in the 1930s, which today is an invaluable tool for computer scientists (described later).

© T J Naughton, Maynooth University, Ireland

Alonzo Church, 1936

- In his 1936 paper *An unsolvable problem in elementary number theory* Church answered Hilbert's third question by showing that there is no decision procedure for arithmetic.
- The numerous interpretations of his paper are now known collectively as *Church's thesis*.

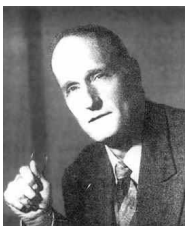
© T J Naughton, Maynooth University, Ireland

Church's thesis, 1936

- Church identified *'effective calculability'* with the operations of his λ-calculus.
- Functions which are *'effectively calculable'* can therefore be evaluated by a purely mechanical process.

© T J Naughton, Maynooth University, Ireland

Stephen Cole Kleene, 1936



- The U.S. mathematician Kleene researched the theory of algorithms and developed the field of recursion theory with Church, Gödel and Turing.
- By providing methods to determine which problems are soluble, Kleene's work led to the study of computable functions and provided the foundations of TCS.

© T J Naughton, Maynooth University, Ireland

Stephen Cole Kleene, 1936

- From the 1930's Kleene more than any other mathematician developed the notions of computability.
- He developed a diverse array of topics in CS, including the computable ordinals, finite automata and regular sets, and recursive realisability with consequences for program correctness.

© T J Naughton, Maynooth University, Ireland

'Effective calculability'

The players

- Gödel at Princeton, 1934, described 'effective calculability' with general recursive functions
- Emil Post, the Polish-American logician, was one of the first to initially bring an idea of physical action into computation, but had not developed his ideas fully by this time

© T J Naughton, Maynooth University, Ireland

'Effective calculability'

The players (cont.)

- Kleene, 1936, fully developed general recursive functions theory
- Church, identified it with λ-functions
- Turing's machines identified computable functions with effective calculability and showed the equivalence of his definition of computability to those of Church and Gödel

© T J Naughton, Maynooth University, Ireland

Church's thesis, 1936

- Turing's thesis (inessential use of)

  Any process that can be naturally called an effective procedure is realised by a Turing machine.
- Church's thesis

  There is an objective notion of 'effective calculability' independent of any particular formalisation.

© T J Naughton, Maynooth University, Ireland

Church's thesis, today

Combining them:

- If any axiomatic-based mathematical construction can solve some problem then a Turing machine can solve that problem.
- Sometimes called the Church-Turing thesis.

© T J Naughton, Maynooth University, Ireland

Church's thesis, today

- The main support for this thesis is that all reasonable formal models of computation that have been proposed (so far!) (the unrestricted models of computation) are provably no more powerful than a TM.
- These formal models of computation are *as powerful as* (and equivalent to) TMs.

© T J Naughton, Maynooth University, Ireland

Unrestricted models of computation

- The λ-calculus - Church, 1936
- General recursive functions - Kleene, 1936
- Turing machines - Turing, 1936
- Post systems - Post, 1943
- Markov systems - Markov, 1954
- Unrestricted rewriting systems (type 0 grammars) - Chomsky, 1959
- RAMs - Shepherdson & Sturgis, 1963
- Multicounter machines - Minsky, 1967
- "Game of life" - Conway, 1970

© T J Naughton, Maynooth University, Ireland

Next we shall take a look at Church's λ-calculus

which extends the power of predicate calculus.

© T J Naughton, Maynooth University, Ireland

Predicate calculus

- A form of logic
  - *a method for describing the form of propositions [e.g. (T^F^T)⇒T] so that it is possible to check in a formal way whether or not a proposition is valid.*
- We use it to express propositions and the relationships between them, and

  to infer new propositions from already established ones using formal rules.

© T J Naughton, Maynooth University, Ireland

Predicate calculus

- We use these terms

  (i) constant symbol - 'Ireland', 'π'

  (ii) variable symbol - 'integer' may be used for all integers

  (iii) compound term - a *function symbol* combined with an ordered set of other terms as arguments

© T J Naughton, Maynooth University, Ireland

Predicate calculus

- A compound term represents some concept
  - likes(cat, cream)
  - root(a)
  - causes(drink(nine), head(sore))
  - (5 ≥ 3)

© T J Naughton, Maynooth University, Ireland

Predicate calculus

- We use logical connectives to construct more complicated expressions from simpler ones
  - negation ¬
  - conjunction ^
  - disjunction ˅
  - implication ⇒
  - equivalence =

© T J Naughton, Maynooth University, Ireland

Predicate calculus

bijection(seqA,**N**)⇒countable(seqA)

- do we mean <u>a</u> sequence called 'seqA' or <u>all</u> sequences?

  Predicate calculus provides us with the *universal* and *existential* quantifiers

  - as seen earlier in set theory.

© T J Naughton, Maynooth University, Ireland

The λ-calculus

- The λ (lambda) calculus was proposed by Church in the early 1930s

  "A set of postulates for the foundation of logic", *Ann. of Math.*, 1932

  as a model for computability and a system for the manipulation of λ-expressions.

© T J Naughton, Maynooth University, Ireland

λ-expressions

- Church, in his original definition, made a distinction between a *denotation* and an *abstraction*.
- The expression (X+1) is used to *denote* a particular number which depends on the value of X.

© T J Naughton, Maynooth University, Ireland

λ-expressions

- Moreover, it can also be taken to express the function inc(X) in which X is *bound*,

  i.e. X must be an actual number, or some function which evaluates to a number,

  rather than a *free* expression (i.e. an untyped variable).

© T J Naughton, Maynooth University, Ireland

λ-expressions

- The expression (X+1) [or the function inc(X)] is also an *abstraction*, because it can apply to many different values of X, not just one value.

© T J Naughton, Maynooth University, Ireland

λ-abstractions

- (λX)M stands for any λ-abstraction in the λ-calculus.
- The λ operator is said to *bind* the individual variable X in the same way that the ∀ and ∃ quantifiers bind an individual variable in the predicate calculus.

© T J Naughton, Maynooth University, Ireland

λ-abstractions

- We shall therefore denote the *increment abstraction* by (λX)(X+1).
- The abstraction (λX)(X+1) also serves as a definition of the *increment function*

  inc(X) = (λX)(X+1).

© T J Naughton, Maynooth University, Ireland

λ-conversion

- If we want to square, say the argument `3' representing the natural number 3, we must use an inference rule called λ-*conversion*.

© T J Naughton, Maynooth University, Ireland

λ-conversion

- This requires us to let S(*a*, X, M) stand for the result of substituting *a* for X in M.

  Here *a* is a free variable which is distinct from the X of S and the bound variables of M.

  In this example M will be (X . X) .

© T J Naughton, Maynooth University, Ireland

λ-conversion

- The internal representation of S

  ((λX)M)(*a*)

  denotes the *application* of the function definition, or λ-abstraction, (λX)M to the argument *a*.

© T J Naughton, Maynooth University, Ireland

λ-conversion

- Now we have

  $((\lambda X)(X \,.\, X)(3) = (3 \,.\, 3) = 9$

  It is also possible in the λ-calculus to use the functional notation, such as that shown previously for the increment definition, so that the application is

  $(\text{square}(X))(3) = (3 \,.\, 3) = 9$

© T J Naughton, Maynooth University, Ireland

The λ-calculus

- In general, we use the λ-calculus to represent *functional* information and to extend the power of the predicate calculus notation.

© T J Naughton, Maynooth University, Ireland

The λ-calculus

- In 1936, Church successfully demonstrated the generality of his λ-calculus, when he formally defined the concept of computability by identifying that which is *'effectively calculable'* with λ-expressions.
- Much contemporary computer program design uses the concept of *functions*.

© T J Naughton, Maynooth University, Ireland

CS605 Computability II