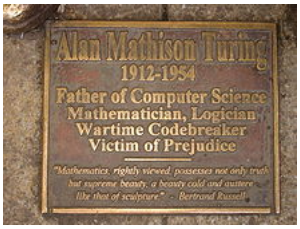
Computability & Uncomputability

Robert Keller
December 2012

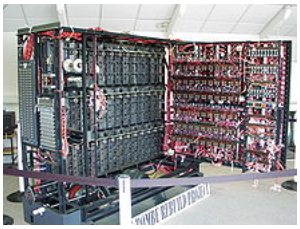
Computability

- What does it mean for a function to be computable?
- Alan Turing (and others) sought the answer to this question ca. 1936.
- Turing's mathematical machine model is generally accepted as capturing the notion of computability.

Alan Turing, 1912-1954



Turing memorial plaque,
Manchester, England

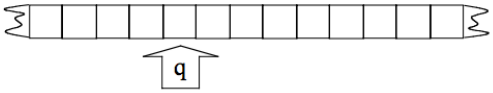


Turing's decryption computer
da Bombe, 1941
which helped win WWII

http://en.wikipedia.org/wiki/Alan_Turing

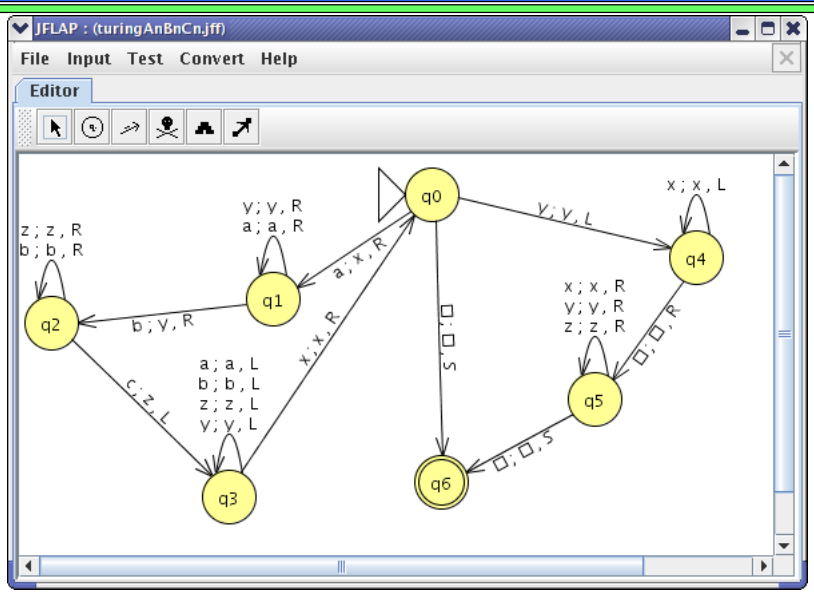
Turing Machines

- I will describe the model on the board.



- I will give a Prolog program that simulates a Turing machine.
- See JFlap for additional examples.
- See also sections 6.5-6.7:
  http://www.cs.hmc.edu/~keller/cs60book/%206%20States%20and%20Transitions.pdf

Turing Machine in JFlap



Things to Remember about Turing Machines

- There is a **finite** set of tape symbols.
- There is a **finite** set of control states (so the control states cannot remember arbitrarily-large numbers).
- Single actions are **local** and bounded in complexity (read, write, move, change-state).
- At any one time, only a **finite portion** of the tape is in use. The size of the portion can increase over time however, so the tape is **unbounded**.
- There is no result until the machine **halts**.

Why accept Turing's concept?

- Turing gave a careful informal argument as to why his machine model captured the essence of computability.
- Other models of computabilty (a dozen or so) are provably equivalent to Turing's machines.
- No one has come up with a function that is intuitively computable that could be proved to be not computable on a Turing machine.

Turing's concept cannot be proved

- In order to **prove** that Turing was correct, we'd have to come with another model, say an **X machine**, for computability that is more obviously correct, then prove that anything computable by an X machine could be computed by a Turing machine.
- But then there would remain the question of whether the X machine completely captured the notion of computability.
- This could lead to infinite regress (X machine, Y machine, Z machine, …).

Turing's idea could conceivably be disproved.

- This could be done by devising a function that is intuitively computable, then proving that it cannot be computed by a Turing machine.
- This has not been done to satisfaction of the general community.
- However, there is work in this direction:
  http://en.wikipedia.org/wiki/Super-recursive_algorithm

Turing computability

- To put arguments about computability on a precise formal basis, one can add the qualification "Turing" in front of "computabilty".
- When we say "computability" here, Turing computability is what we mean.

Programming Languages

- Functions that are computable in most ordinary programming languages (Racket, Java, C++, C, Basic, …) are computable by Turing machines, and vice-versa.
- We assume that the platform on which programs in these languages executes has unbounded memory available (just like a Turing machine tape does).

Mutual Simulation

- It is relatively easy to show that all common programming languages can simulate an arbitrary Turing machine.
- It is also possible, but generally tedious, to show that a Turing machine can simulate programs in common languages.

Why do we care?

- In showing that there are functions that are not computable, it may be easier to use a programming language.

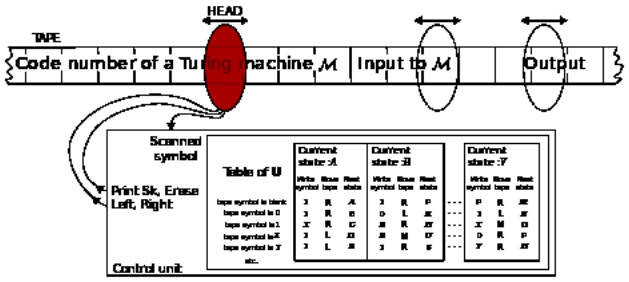
Universal Turing machines

- A universal Turing machine is one can simulate any other Turing machine.
- The alphabet of a universal TM is fixed.

  So we'd need to encode the alphabet of the TM being simulated.
- The program of the simulated TM can be arbitrarily large.

  So we'd need to encode and store the program of the simulated TM on the universal machine's tape.
- The universal machine would "jockey back and forth" between the program and the simulated tape, leaving markers, etc. in the form of tape symbols to keep track of current state and head position.

Universal Turing machine



http://en.wikipedia.org/wiki/Universal_Turing_machine

Most functions are not computable

- Surprised?
- Suppose computability is defined to be computable by a TM, or by some language.
- There is a countably-infinite set of strings over a finite alphabet, thus a countably-infinite set of programs.
- So the set of computable functions (on, say, the natural numbers) is countably-infinite.

What does “countably infinite” mean?

- It means that the set can be put into 1-1 correspondence with the natural numbers {0, 1, 2, 3, …}.
- For example, these are countably infinite:

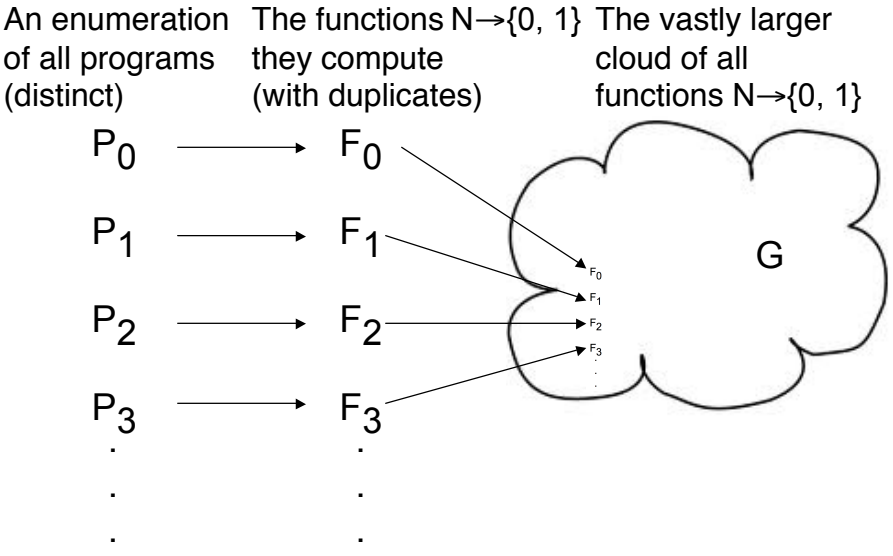
  The set of all pairs of natural numbers:
  {(0, 0), (0, 1), (1, 0), (0, 2), (1, 1), (2, 0), …}

  The set of all finite lists of natural numbers

The set of computable functions

- Consider just the functions of the form $N \rightarrow \{0, 1\}$.
- Again, the set of programs available to compute these is countably-infinite.
- But the set of such functions is ***not*** countably-infinite. There aren’t enough programs to “go around” for all of the functions.

Programs vs. Functions



Set of functions N → {0, 1} is uncountable

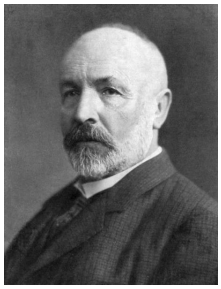
- This observation was made by Georg Cantor, in his famous **Diagonal Argument**, 1891.
- Suppose to the contrary that the set of such functions **is** countable:
  $F_0, F_1, F_2, \ldots$
- Define "diagonal" function G such that
  $\forall n\ G(n) = 1\text{-}F_n(n)$.
- G is clearly a function in the set N → {0, 1}.

Set of functions N → {0, 1} is uncountable

- Define “diagonal” function G such that
  $\forall n\ G(n) = 1\text{-}F_n(n)$.
- But G can’t be in the list as F_m for any m. Why?
- If G were, say, F_m then $F_m(m) = G(m) = 1\text{-}F_m(m)$.
- But $F_m(m) = 1\text{-}F_m(m)$ is impossible for functions having values in {0, 1} only.

David Hilbert’s comment on Cantor

"No one shall expel us from the paradise that Cantor has created."



Georg Cantor,
1845-1918

Constructing Specific Uncomputable Functions

- To demonstrate specific uncomputable functions, we'll use a programming language, Racket.
- First we'll get comfortable with the notion of "self-applicability":

  applying a function to its own program.

Self-Application

Functions are applied to themselves on a routine basis:

A program that **counts characters** in a file can count the number of characters in its own source code.

A **word processor** can read and format its own source code.

A **compiler** (program that translates language statements to machine language) can be written in the language that it compiles.

Self-Application in Racket

- (lambda(x) 999) returns 999 for any argument
- Here are example applications:

  ((lambda(x) 999) 0) returns 999
  ((lambda(x) 999) ‘foo) returns 999
  ((lambda(x) 999) ‘(foo bar)) returns 999

- What does
  ((lambda(x) 999) ‘(lambda(x) 999))
  return?

Self-Application in Racket

- What does
  ((lambda(x) x) ‘(lambda(x) x))
  return?

Evaluating a Literal Program

- We know that the built-in **eval** will evaluate any expression in Racket, e.g.

  (**eval** ‘(+ 2 3))

  gives the same result as (+ 2 3).

eval1

- Purely for convenience, we define **eval1** that evaluates the application of a quoted lambda expression of 1 argument to its argument.

- Example:

  (**eval1** ‘(lambda(x)(* x x)) ‘5)

  gives the same result as

  ((lambda(x)(* x x)) ‘5)

  i.e. 25.

eval1

- (define (**eval1** P I)
    (eval (list P I)))
- (list P I) makes an application (P I)
- (eval (list P I)) evaluates that application

Divergent Applications

- Consider an application (P I).
- This application could return a value.
- Or it might **diverge** (return nothing).

Uncomputable Functions

- **There are functions that Racket cannot compute.**
- As Racket is a universal language (capable of simulating any Turing machine), there are functions that no programming language can compute.

A Specific Uncomputable Function

- A universal diagnosing function: **diverges?**
- It would be nice to have a function that tells whether an arbitrary function implementation diverges on a given input, just by examining its source code.
- Given this function, we could determine whether any implementation of a function of interest has a bug.

diverges?

- The meaning of the function **diverges?** is

  (**diverges?** P I) returns #t provided
  P is a 1-argument lambda expression
  and (P I) diverges.

  Otherwise (**diverges?** P I) intentionally diverges.

Note that **diverges?** is the same type as **eval1**.

Intentionally diverge?

- It is easy to *intentionally* **diverge**.
- (define (**diverge** x) (**diverge** x))

or without using define:

- ((lambda (x) (x x)) (lambda (x) (x x)))

A form **diverges?** might take

```
(define (diverges? P I)
    (if (and
            (is-lambda-expression-of-1-arg? P)
            … other conditions …
        )
    #t                  ; return #t
    (diverge I)         ; diverge on purpose
)
```

A feeble attempt

```
(define (diverges? P I)
   (if
      (and
            (lambda-expression-of-1-arg? P)
            (> (length (third P)) 999)
      )
    #t
    (diverge I)
)
```

Obviously the first conjunct is computable.

It will take more than the second conjunct to give a correct answer, of course.

The point is: such a program can be executed.

Possible behaviors of diverges?

- By definition, there are only 2 possible behaviors on a given input:

  diverges (never gives result)

  returns #t

Why **diverges?** can't be computed

- No matter how sophisticated we make the test in **diverges?**, it will never do its intended job correctly.
- Suppose **diverges?** were computable.
- Consider **(lambda(P) (diverges? P P))** (*)
- This is a valid function expression (could be defined as **self-diverges?**).
- If **diverges?** is computable, so is (*).

Self-Apply

- Apply this:

  (lambda(P) (diverges? P P)) (*)

- To this:
  ‘(lambda(P) (diverges? P P))

- **((lambda(P) (diverges? P P))**
  ‘(lambda(P) (diverges? P P))) (**)

- What is the result of (**)?

Evaluating

- By substituting for P in (**):
  ((lambda(P) (diverges? P P)) ‘(lambda(P) (diverges? P P)))

- the result of (**) is precisely that of

  (diverges? ‘(lambda(P) (diverges? P P))
  ‘(lambda(P) (diverges? P P))) (***)

- which, by definition of diverges? must be one of
  #t
  divergent (i.e. no result)

Case (***) evaluates to #t

If

(diverges? '(lambda(P) (diverges? P P))
'(lambda(P) (diverges? P P))) (***)

evaluates to #t, this says that

(lambda(P) (diverges? P P)) (*)

diverges when applied to '**(lambda(P) (diverges? P P))**.

But the latter application ***is*** (**):

((lambda(P) (diverges? P P)) '(lambda(P) (diverges? P P)))

the very thing that **diverges?** says **evaluates to #t**.

No application can both evaluate to #t and diverge; the two behaviors are mutually exclusive.

So this case leads to a **contradiction**.

Case (***) diverges

If

(diverges? '(lambda(P) (diverges? P P))
'(lambda(P) (diverges? P P))) (***)

diverges, this says that

(lambda(P) (diverges? P P)) (*)

returns #t when applied '**(lambda(P) (diverges? P P))**

Again, that application is just (**):

((lambda(P) (diverges? P P)) '(lambda(P) (diverges? P P)))

But assuming **diverges?** does its job, returning #t says that the application (**) diverges.

Again, no application can both evaluate to #t and diverge; the two behaviors are mutually exclusive.

So this case also leads to a **contradiction**.

No Smoke and Mirrors

- **All the expressions are valid Racket:**

  (define (**diverges?** P I) …)

  ((lambda(P) (**diverges?** P P)) '(lambda(P) (**diverges?** P P)))

- **The use of define is dispensable:**

  (lambda (P I) …)

  can replace **diverges?**

Unsolvable Problems

When we speak of a “solving a problem” we mean constructing an **program** that will determine whether there is a solution to an **arbitrary instance** of the problem.

Example of a Problem

- **Context-Free Grammar Fullness:**
  Given a context-free grammar over alphabet Σ, does the grammar generate every string in Σ^*?
- An **instance** of the problem is a specific context-free grammar.
- The **problem** itself is whether there is a program that will answer for any instance.

Solvability of a Problem

- A problem is called **solvable** if there is a program that will determine the answer for any instance.
- The context-free grammar fullness problem is an example of an **unsolvable problem**.
- In contrast, the finite-state machine fullness problem is solvable.

Unsolvability

- "Unsolvable" does not mean "not yet solved". (The word "open" is used for that.)
- "Unsolvable" means "cannot be solved".

Example: Post's Correspondence Problem

- An **instance** of the problem is:
  Given a finite set of pairs of strings (x_i, y_i) over an alphabet
  is there a sequence i, j, k, … n,
  where some indices may repeat,
  such that $x_i\, x_j\, x_k \ldots x_n = y_i\, y_j\, y_k \ldots y_n$
- The problem is to **construct a program** that will solve any instance.
- This is another **unsolvable** problem.

Example of PCP

- Consider the pairs
  (a, baa), (ab, aa), (bba, bb).
- This instance has a solution 3,2,3,1:

  (bba, bb) (ab, aa)(bba, bb)(a, baa)

  because (when blanks are ignored):

  bba ab bba a = bb aa bb baa

Example of PCP

- Consider the pairs
  (a, baa), (ab, aa), (bba, ba).
- This instance is has no solution.

Unsolvability of “The Halting Problem”

- The Halting Problem is to construct a function (**halts?** P I) = #t if P is a program that halts on input I, and #f otherwise.
- If we could construct **halts?**, we could construct **diverges?** as follows:
- (define (**diverges?** P I)
  (if (**halts?** P I)
  (diverge I) ; intentionally
  #t ; return #t if (P I) diverges
  ))
- Thus **halts** is uncomputable too.

The **diverges?** Fountain

diverges? is the real fountain from which other unsolvable problems spring.

More Uncomputable Functions

Consider the 1-argument function:

(**diverges-empty?** P)

is #t if P diverges on '()

diverges otherwise.

This function is also uncomputable, even though it looks simpler than **diverges?**

Proof by Reduction

- We show that if **diverges-empty?** were computable, then so would **diverges?** be.
- But since we already showed **diverges?** is not computable, **diverges-empty?** cannot be either.
- **Caution**: The argument here is a lot more subtle, as it involves construction of programs on the fly.

Proof by Reduction

- We give a definition of **diverges?** that uses the assumed-computable **diverges-empty?**

```
(define (diverges? P I)
  (let (
        (Q (list 'lambda '(J) (list P I)))
        )
   (diverges-empty? Q)
 ))
```

Construction

- Given P and I, the program constructs a quoted lambda expression of the form:

  ‘(lambda (J) (P I))

- If applied to an input, this program would just ignore the input and behave exactly as P does on I.

What happens here?

- Given a arguments P I, (**diverges?** P I) **constructs** a new program Q.
- Q with input J ignores J and behaves as P on input I.
- So Q on input '() gives the same result, if any, as P would on input I.
- Thus if **diverges-empty?** were computable, so would **diverges?** be. But we know the latter is not.

Execution Note

- The programs being constructed are for analysis purposes.
- The analyzer may or may not ever try executing them (e.g. with eval1).

Similar Reductions

- These functions are shown uncomputable in similar ways to the preceding.

  diverges-X? where X is *any* **fixed** input

  diverges-all? meaning the program argument diverges on **all** inputs

  diverges-some? meaning the program argument diverges on **some** input

Computable functions

- (**semi-halts?** P I):
  If P halts on I, then return #t,
  else diverge

- (define (**semi-halts?** P I)
  (begin (**eval1** P I) #t)
)

This function actually *runs* P, but most preceding constructions do not necessarily.

Computable functions

- (**halts-some?** P I):
  If P halts on *some* input, then return #t,
  else diverge

- How could this function be constructed (informally)?

Uncomputable

- **halts-all?** If P halts on all inputs, divergent otherwise

Although this is by now intuitively uncomputable, showing it is another matter.

The proof will be reserved for CS 81.

Rice's Theorem

- If X is a *property of the function computed* by a program P (in the sense that if two programs that compute a given function either both have X or neither does), *and*
- X is non-trivial (meaning some programs' functions have X and some don't),
- then there is no program that can determine whether or not an arbitrary program has X.

Proof

- Let X be a non-trivial functional property.
- We want to show there is no computable function that determines X of a program, yes-or-no. If there were, we could construct halts?
- If function **diverge** has property X, then continue the discussion replacing X with not-X.
- So proceed knowing **diverge** does not have property X.

Proof Continued

- Let M be some fixed program that *does* have X.
- To construct **halts?** from a program **hasX?** that determines property X:

```
(define (halts? P I)
    (let (
        (Q (list 'lambda '(J)
            (list 'begin (list 'eval1 P I)
                         (list 'eval1 M J))))))
    (hasX? Q))
```

Proof Concluded

- Suppose P and I are supplied to this version of **halts?**.
- Then Q is constructed, which looks like (lambda (J) (begin (eval1 P I) (eval1 M J)))
- If P halts on I then the result of (Q J) is the same as (M J), i.e. Q has the same properties as M.
- If P does not halt on I, then (Q J) diverges.
- So Q behaves **as M** if P halts on I, and as **diverge** if P does not halt on I.
- In other words, Q **has X** if P halts on I and **does not have X** if P does not halt on I (recalling that **diverge** does not have X).
- So if **hasX?** could determine whether Q has X, it can be used to construct **halts?**
- Therefore **hasX?** *cannot* be constructed.

Where Rice's Theorem Does Not Apply

- Rice's theorem only applies to **functional** properties X, not **structural** properties.

- Examples of non-functional properties:

  Does a program have more than 1000 lines?

  Does a program always halt in less than 1000 steps?

Complexity Hierarchies

- Functions can be classified by computational difficulty, such as:

- Computable in linear time (linear in size of input).
- Computable in polynomial time.
- Computable in exponential time.

  . . .

- Uncomputable
- Uncomputable of higher degrees . . .

Language Hierarchies

- Finite-state machines: Regular or type 3 languages
- Pushdown automata: Context-free or type 2 languages
- Linear-bounded automata: Context-sensitive or type 1 languages
- Turing machines: Type 0 languages