CS 3100, Models of Computation, Spring 20, Lec 23
April 6, 2020

Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, UT 84112

URL: https://bit.ly/3100s20Syllabus

SCHOOL OF COMPUTING
THE UNIVERSITY OF UTAH

Agenda for Wed Apr 6

- Review of Functions
  - Review of basic concepts such as Domain, CoDomain, Range, 1-1, Onto
- Cantor-Schroder-Bernstein Theorem (CSB Theorem)
  - Counting sets using the SBT
  - Show that there are as many TMs as Nat
- Showing that there exist non-RE languages (Appendix C)
  - Show that a Correspondence between TMs and Languages FAILS
  - Hence non-RE sets exist
- Mapping Reductions: Two Approaches
  - Solver for New Problem helping solve an Already Impossible Problem
  - As a transformation from an Old Impossible problem to a New Problem

Review of Functions

Review of Functions

- Functions map Domains to their CoDomains
- Where the mapped points fall is called the Range of a function
- Functions can be 1-1 (injections) or Many-to-one
- Functions can be Onto (covers the range) or Into (does not)
- Which of these are 1-1 and which are many-to-one ?
  - Rot13: {A,B,C, …, Z} ➔ {A,B,C,…,Z} where rot13 maps each character to the 13th next character; for instance
    - A <-> N, B <-> O, …., L <-> Y, M <-> Z
  - Mod3 : Nat ➔ Nat where
    - 0 -> 0, 1 -> 1, 2 -> 2, 3 -> 0, 4 -> 1, …

Review of Functions

- Is Rot13 Onto, where Rot13: {A,B,C, …, Z} ➔ {A,B,C,…,Z} ?
- Mod3 : Nat -> Nat Onto?
- Suppose we define Mod3 : Nat ➔ {0,1,2} then is Mod3 Onto?

- Functions that are 1-1 and Onto are important
  - They are called Bijections

- Establishes a 1-1 Correspondence between the sets
  - E.g. a "barter"

Fun Exercise using Functions: the CSB Theorem

- We can "count" infinite sets via a Correspondence
- I.e. we place the sets into a 1-1, Onto arrangement (Correspondence)
  - Example : Nat ➔ Int
  - Recall Nat = {0,1,2,3…}
  - And Int = {0, 1, -1, 2, -2, 3, -3, …}
- Suppose we specify this function: h : Nat ➔ Int
  - 0 -> 0, 1 -> 1, 2 -> -1, 3 -> 2, 4 -> -2, 5 -> 3, 6 -> -3
  - Odd n -> (n+1)/2 Even n -> - n/2
- This is a Correspondence between Nat and Int
- There are as many Nat as Int -- even though Nat is a proper subset of Int !!

Fun Exercise using Functions: the CSB Theorem

- This is a 1-1 Correspondence between Nat and Int
- There are as many Nat as Int -- even though Nat is a proper subset of Int !!

- The cardinality of Nat is Aleph_0
- Hence the cardinality of Int is also Aleph_0
  - Because there is a 1-1 correspondence between Nat and Int

Fun Exercise using Functions: the CSB Theorem

- Finding a 1-1 correspondence between Nat and Int was easy
  - Odd n -> (n+1)/2 Even n -> - n/2
- But often it is not that easy
- E.g. how do we find a 1-1 correspondence between
  - Nat and C programs ?
  - Nat and TMs ?
- The Cantor-Schroder-Bernstein Theorem helps find a 1-1 correspondence by merely asking you to find 1-1 maps going both ways!

The Cantor-Schroder-Bernstein Theorem

The Cantor-Schroder-Bernstein Theorem

- For sets A and B

- If there is a 1-1 map f : A -> B
- And there is a 1-1 map g : B -> A

- Then there is a 1-1 Correspondence h : A -> B
- i.e. we will now have a function "h" that is a 1-1 and Onto map from A and B
  - And by virtue of that, the inverse of h must exist also !!

Illustration of the CSB Theorem

- **Show that** there are Aleph_0 C programs (same cardinality as Nat)
- SBT requires two 1-1 maps
  - One from Nat to C and the other from C to Nat
- Nat to C : 0 -> main(){} , 1 -> main(){;}, 2 -> main(){ ;; }, etc
  - All these trivial C programs are legal; they do compile and run!
  - We don't need to hit all C programs! Just finding ANY 1-1 map is sufficient
- C to Nat : just take the ASCII string and read it as a Nat ☺
- **Hence proved!** We just proved that there are as many C programs as Nat
- The same argument can be applied to TMs
  - Take 0 -> First-Trivial-TM , 1 -> First-Trivial-TM-with-One-NoOp-move
  - Take 2 -> First-Trivial-TM-with-Two-NoOp-moves, etc.

There exist non-RE Languages

- There are as many RE languages as Turing Machines
  - Because each TM goes with an RE language
- Thus there are Aleph_0 RE languages

- We will now show that there are Aleph_1 languages
  - The cardinality of Languages is Aleph_1
  - Aleph_1 is the cardinality of Reals

- Thus there are more languages than RE languages
  - Or some language is non-RE !!

Now to show there are ”more languages”

- Suspose there is a 1-1 Correspondence between TMs and languages

	‘’	0	1	00	01	10	11	000	001…. (all possible strings)
TM0	0	1	0	0	1	1	0 …		→ language {0,01,10}
TM1	0	0	0	….					-> Language { }
TM2	1	1	1	1					-> Language Sigma*
TM3	1	0	1	0	1	0	… ->		-> Alternate strings in num order

The above listing shows all candidate strings on the top row

And uses a “bit vector” to pull out languages

Here we portray as if TM0’s language is {0, 01, 10

TM1’s language is portrayed as { }

TM2’s language is portrayed as Sigma*

TM3’s language is portrayed as {‘’, 1, 01, …. (alternate strings of the numeric order) }

Now to show there are ”more languages”

- Then in this listing, consider the COMPLEMENT of the diagonal

	‘’	0	1	00	01	10	11	000	001.... (all possible strings)	
TM0	0	1	0	0	1	1	0 ...			→ language {0,01,10}
TM1	0	0	0	....						-> Language { }
TM2	1	1	1	1						-> Language Sigma*
TM3	1	0	1	0	1	0	... ->			-> Alternate strings in num order

Then the DIAGONAL LANGUAGE – the language obtained by complementing the diagonal CANNOT be in the listing because it differs from each listed language at least at one place

But since there are Aleph_0 TMs, there must be Aleph_1 languages

(there is no cardinality between Aleph_0 and Aleph_1)

And in fact we can read out each “bit vector” for a language as a Real Number also !!

Mapping Reductions

Basics of Mapping Reductions

- Mapping Reductions help “bridge” from an OLD and HARD problem to a potentially HARDER problem

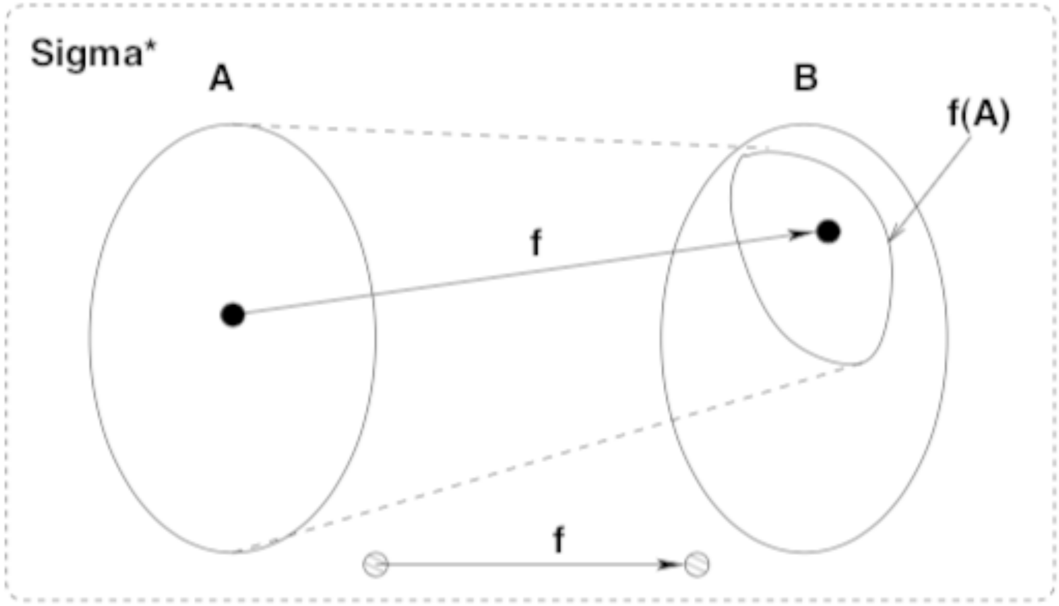
- This way if the HARDER problem can be solved, then the OLD and HARD problem can be solved

- This is how we show that TMs ”can’t solve certain problems”
- This is how we show that some problems are NP-complete

Mapping Reductions: Definition

Mapping Reduction : Definition w.r.t. Language Mapping

- A language A is mapping-reduced to a language B via function f
  - Written A <=m B
  - If the following holds
  - We find a Turing-computable function “Translate” (“f” below) such that
  - For any x in Sigma*
    - x in A iff Translate(x) in B

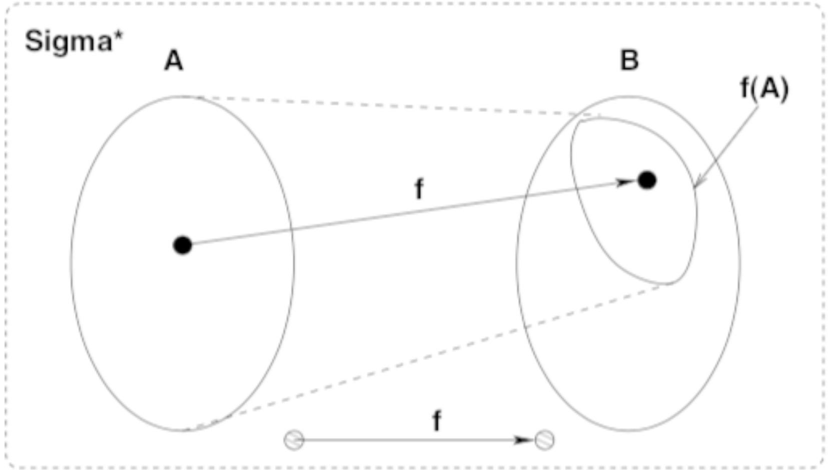


Mapping Reduction : Definition w.r.t. Problem-Solving

- Problem A is mapping-reduced to Problem B via function f
  - Written A <=m B
  - If the following holds
  - We find a Turing-computable function “Translate” (“f” below) such that
  - For any x in Sigma*
  - We produce f(x) also falling into Sigma*

  - But we focus on what x does to points in A
  - Those points must fall INTO B

  - Points x outside of A must fall OUTSIDE B



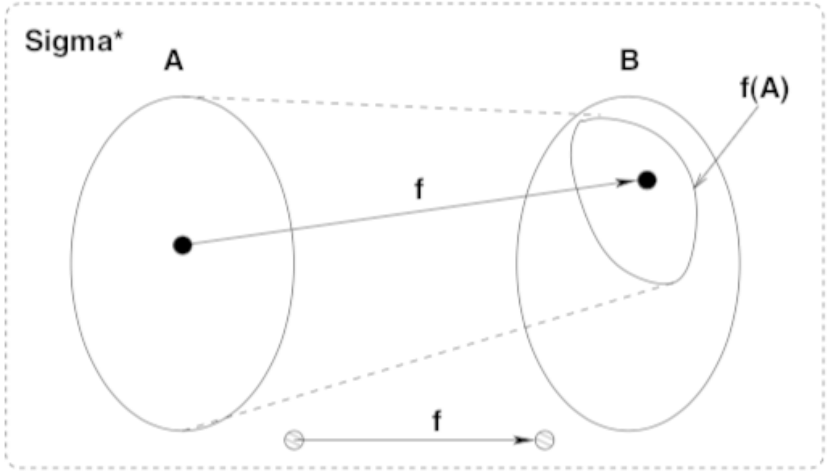
Mapping Reduction : Definition w.r.t. Problems

- Problem A is mapping-reduced to Problem B via function f
  - The ENTIRETY of A is mapped into a subspace of B via f
  - A are “all the Old and Hard” problem instances
  - B are the “New and Potentially Harder” problem instances

  - Thus, if there is an algorithm for B
  - We must have an algorithm for the f(A) region which is contained in B

  - But we now have an algorithm for A also!
    - Given an instance x in A, map it via “f” into B
    - Then solve the mapped problem via B’s algorit

  - Let us illustrate these specifically on an example



Mapping Reductions: Examples

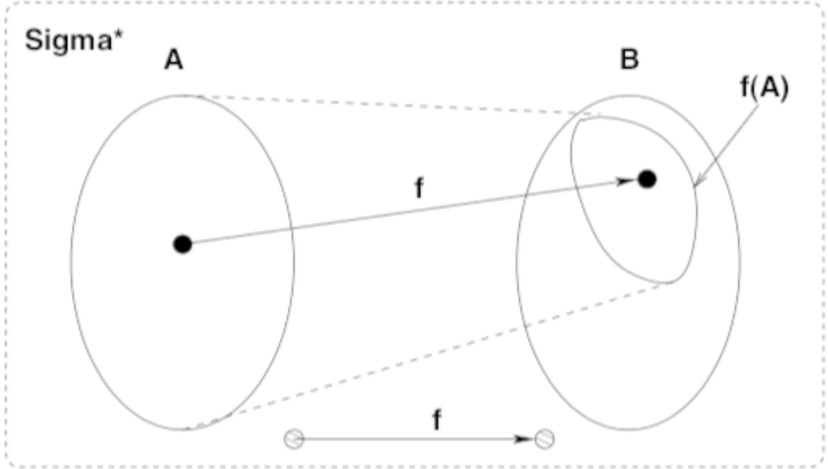
Regular_TM is not decidable (has no algorithm)

- Regular_TM is a language
- Regular_TM = { <M> : The language of M is regular }
- Suppose Regular_TM is decidable
- Then we can build a decider for A_TM

- But we have already shown that A_TM does not have an algorithm
- Thus, we cannot have an algorithm for Regular_TM (Reg_TM)

- We will show A_TM <=m Reg_TM by presenting the “f” function

The Mapping Reduction Picture now



- Let “A” be A_TM
- Let “B” be Reg_TM
- We want a function “f”
- f must be a computable function
- Thus, f is a program-function that translates an <M,w> within A_TM into an M’ that falls into Reg_TM

We will define such a translator called “Translate” (next slide)

Proof that Regular_TM is not recursive: define function “Translate” as follows

```
CProg Translate(CProg M, string w) { // This is the mapping reduction function
 printf
   ("
    M’(x) {
      if x is of the form 0^n 1^n then goto accept_M’;

      Run %s1 on %s2 ; // First %s1 will splice-in M, second %s2 will splice-in w

      If this execution results in %s1 accepting %s2,
      then M’ goes to accept_M’;

      If %s1 rejects %s2, then M’ goes to reject_M’;

    }", M, w);

}
```

What does Translate yield?

- Translate yields the description of a TM M’
- That is what the “print” produces

- Now suppose FullDeciderRegTM exists (algorithm for Reg_TM)

- How can we build FullDeciderATM ? (algorithm for A_TM)

FullDeciderATM is built as follows

```
CProg FullDeciderATM(CProg M, string w) { // This is the mapping reduction function

  return FullDeciderRegTM( Translate(M,w) ) ;

}
```

What are the strings that fall into the language of M’ ??

- First they are strings “x” of the form 0^n and 1^n

- Then there are additional strings (all other strings in fact), provided M accepts w

- What is the language of M’ if M accepts w?
- What is the language of M’ if M does not accept w?
- When is it that the language of M’ is regualr ??

Does Reg_TM have an algorithm?

- There are only two possibilities:
  - Either M’ has a regular language or not

- We see that M’ includes strings of the form 0^n 1^n always

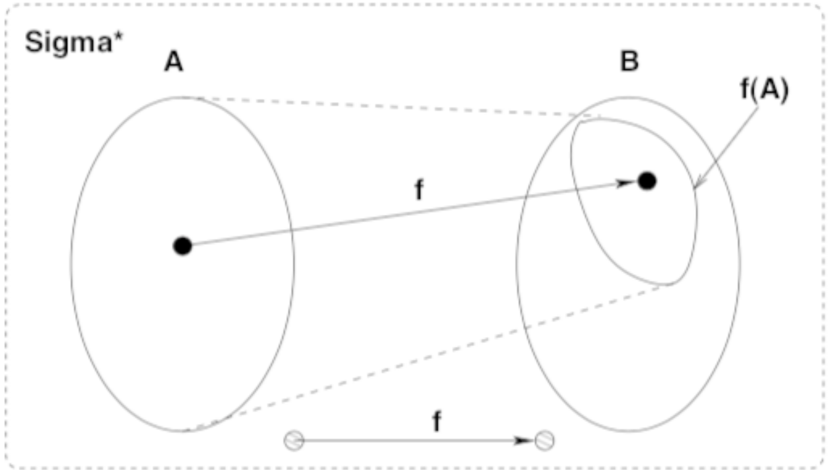
- But if M accepts w, then M’ also includes strings that are NOT of the form 0^n 1^n

- Thus if M accepts w, the language of M’ is regular
- If M does not accept w, the language of M’ is not regular
- Thus we have shown A_TM <=m   Reg_TM

Asg-6 asks you to modify this proof for CFL_TM

Asg-6 also asks you to show Amb has no algo.

- Let “A” be PCP
- Let “B” be AMB
- Function f is given to you
- f takes a collection of tiles (PCP inst.) and produces a CFG



Such that the CFG will be unambiguous IF and ONLY IF the PCP system has no solution.

Thus if we bring along an algorithm for Amb, we will have an algorithm for PCP

Thus we have shown PCP <=m Amb

Ways of pictorically presenting <=m

Boxes inside boxes view of A_TM to Reg_TM

If you have a Reg_TM decider, you can build up an A_TM decider which is “known impossible”