End of Undecidability; Start of Boolean Logic

CS254

Chris Pollett

Sep 20, 2006.

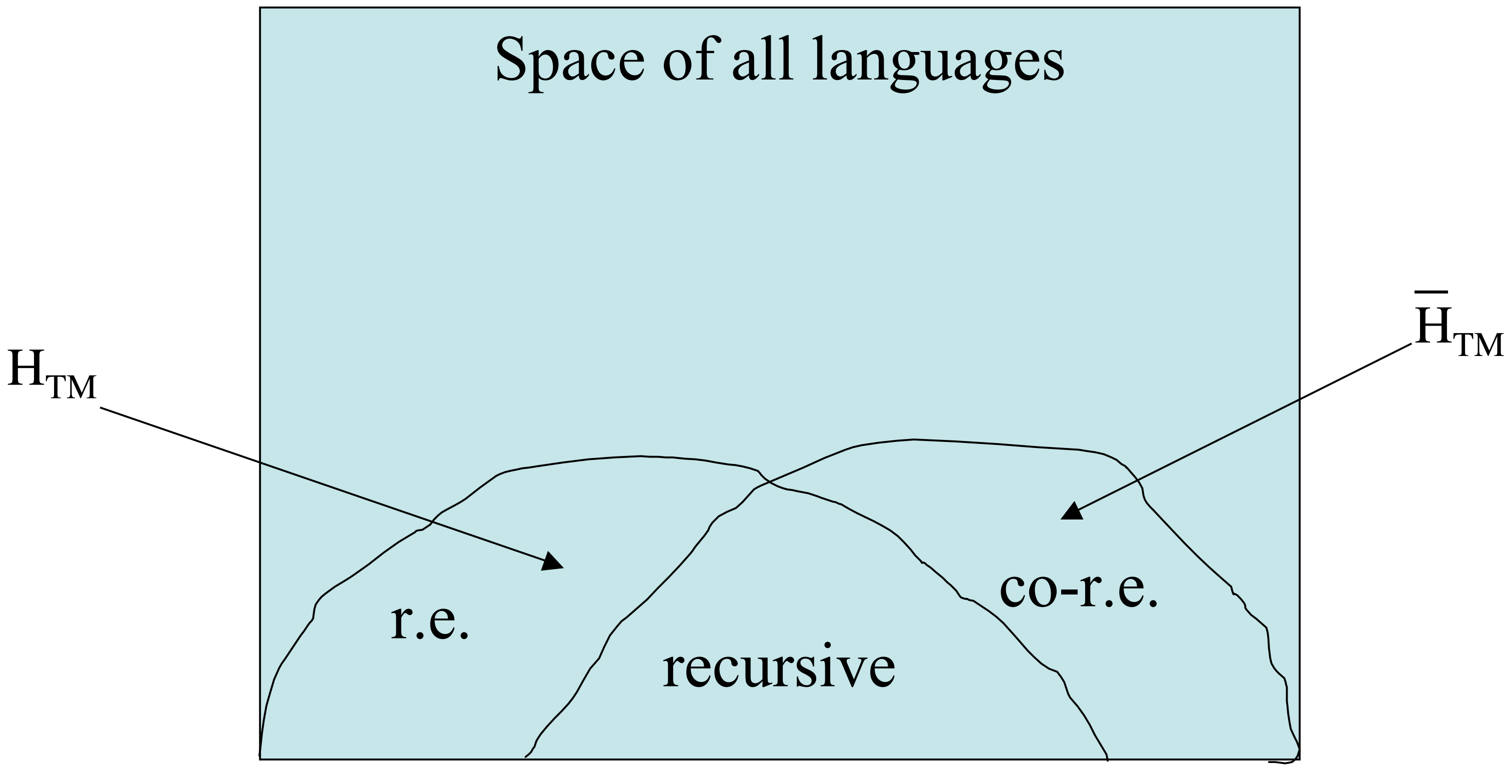
Outline

- R, r.e., co-r.e. structure
- Enumerators
- Rice’s Theorem
- Boolean Logic

Summary of Structural Results

- Let co-r.e. denote the languages whose complement is an r.e. language.
- The set of language which are either r.e. or co-r.e. is countable, therefore by last day we know that there are languages which are neither r.e. nor co-r.e.
- Since the complement of any recursive language is recursive, we know $\overline{H}_{TM}$ is not recursive.
- We also know from last day that if L is r.e. and co-r.e then L is recursive.
- So we have established our first structural diagram for languages…

Structure Diagram



Enumerators

- An **enumerator** is a Turing machine which starts on a blank tape and starts computing. (It is where the term recursive enumerable comes from)
- The language output by an enumerator E is the set of strings x such that at some point in E's computation the input tape looked like y_x_.

Proposition. L is recursively enumerable iff it is the language of some enumerator. (Remark: iff is an abbreviation for if and only if)

Proof. Suppose L is recursively enumerable by a 1-tape M. Consider the enumerator which has three tapes. The second tape keeps track of the stage; the third take keeps track of the string. In stage i the enumerator, cycles through each of the lexicographically first i strings in the alphabet (on tape 3), writes one to the input tape and simulate M for i steps on this string. If M accepts it erases the input tape and writes this string there. This shows one direction. Suppose L is enumerated by some E. Consider the machine which on input x, simulates E on auxiliary tapes step by step, after each step it checks: has E output x? If it has, the the machines halt in state yes.

Rice’s Theorem

- This theorem shows that almost any problem one could come up with connected to Turing Machines is undecidable.

Theorem. Let P be a language such that there exists TM descriptions $<M> \in P$ and $<M'> \notin P$. Further assume that whenever we have two machines M_1 and M_2 such that $L(M_1) = L(M_2)$, then we have $<M_1> \in P$ iff $<M_2> \in P$. Then P is undecidable.

Proof. Suppose we had a decider R for P. We show how to use R to build a decider for H_{TM}. Let $T_\emptyset$ be a TM which never halts. We may assume $<T_\emptyset> \notin P$; otherwise, we carry out our argument using $\overline{P}$. Because P is not trivial there exists a TM T wit $<T> \in P$. Using these machines consider the following decider S for H_{TM}:

S = “On input $<M, w>$:

1. Use M and w to construct the following TM M_w:

   M_w = “ On input x:

   1. Simulate M on w. If it halts proceed to stage 2.
   2. Simulate T on x.”

   So if M halts on w M_w has the same language as T; otherwise, S has the same language as $T_\emptyset$.

2. Use TM R to determine whether $< M_w > \in P$. If yes, *accept*. If no, *reject*.”

Example Use of Rice's Theorem

- Consider the language L={<M>| L(M) contains the string 01}
- Then the machine M_1 which immediately halts in the 'no' state is not in the language
- The machine M_2 which accepts just the string 01 is in the language.
- Notice further if we have two machines with the same language, that language will either have 01 or not. So their codes will either both be in L or both not in L.
- So Rice's Theorem applies and we can conclude L is not recursive (i.e., L is undecidable).

Boolean Logic

- Logic is closely related to computation.
- Over the next couple lectures we will explore this connections as well as the basics of Boolean logic

Boolean Expressions

- Are built out of a countable set of variables $X=\{x_1,x_2, \ldots\}$, and the operations AND ($\wedge$), OR ($\vee$), and NOT ($\neg$) as follows:

Defn. A *Boolean expression* can be any one of (a) a Boolean variable, (b) $\neg F$ provided F is a Boolean expression, (c) $(F \wedge G)$ provided F, G are a Boolean expressions, or (d) $(F \vee G)$ provided F, G are a Boolean expressions. Case (b) is called the *negation* of F; case (c) is called the *conjunction* of F and G; and case (d) is called the *disjunction* of F and G.

- For example, $F := (\neg(x_1 \vee x_2) \wedge x_5)$ is a Boolean expression.

Truth Assignments

- A *truth assignment* is a mapping T from a finite subset X´ of variables X to the set {true, false}
- We define T *satisfies (or models) a formula* F, written T |= F, inductively. If F is a variable then T |= F means T(F) = true. If F is of the form ¬G, then T|= F holds provide T did not satisfy G . That is, T|=/=G . If F is of the form (G ∧ H) then T|=F holds provided both T|=G and T|=H hold. Finally, if F is of the form (G ∨ H) then T|=F provided at least one of T|=G or T|=H hold.
- For example, consider the formula F of the last slide. Let $T(x_1)=T(x_2)=T(x_3)$ = false, $T(x_4) = T(x_5)$ = true, then T |= F.
- By considering different truth assignments we can view this F as a function from three boolean variables to true or false.
- We write F=>G as an abbreviation for (¬F ∨ G) and we write F <=> G as an abbreviation for (F=>G) ∧(G=>F)
- We say two formulas F, G are equivalent, written F≡G, if for any T, T|=F iff T|=G.
- Proposition 4.1 in the book gives a list of common equivalences among boolean expressions, for instance, things like (F ∨ G) ≡(G ∨ F) and DeMorgan's laws. You should know these.