1 Warm Up (#Computability, #Logic)

1.1 Show that both the collection of Turing-decidable languages and the collection of Turing-recognizable languages are closed under concatenation.

Turing-decidable languages A Turing-decidable language is a language that is decided by some Turing machine.

To prove this we can use proof by construction. Let's assume we have two decidable languages L_1 and L_2 that are decided by the two TMs T_1 and T_2 respectively. Now we can construct a new TM T' that will decide the language $L' = L_1 \circ L_2$.

T' = "On input w:

1 For $i = 0, 1, 2 \ldots$

2 Split w into $w_{part1} = w_1 \ldots w_i$ and $w_{part2} = w_{i+1} \ldots w_n$, where n is the length of the input string.

3 Run T_1 on w_{part1}. If T_1 rejects, reject. If it accepts, go to next step.

4 Run T_2 on w_{part2}. If T_2 rejects, reject. If it accepts, go to next step.

5 Otherwise, accept."

To demonstrate how this works let's use an example. Let $L_1 = a*$ and $L_2 = b*$. Let the input string be $w = ab$. Now let's see how T would decide it. On the first step $i = 0$, $w_{part1} = \varepsilon$ and $w_{part2} = ab$. So T_1 would not reject, but then T_2 would so T' rejects. The next time the process goes $i = 1$, $w_{part1} = a$ and $w_{part2} = b$. So T_1 would not reject a, and then T_2 would also not reject b, so T' accepts.

Turing-recognizable languages For Turing-recognizable languages we can follow the same proof by construction. The only difference here is that sometimes either T_1 or T_2 can be stuck looping, which is not a problem since we are constructing a recognizer T'. So again, the construction is:

T' = "On input w:

1 For $i = 0, 1, 2 \ldots$

2 Split w into $w_{part1} = w_1 \ldots w_i$ and $w_{part2} = w_{i+1} \ldots w_n$, where n is the length of the input string.

3 Run T_1 on w_{part1}. If T_1 rejects, reject. If it accepts, go to next step.

4 Run T_2 on w_{part2}. If T_2 rejects, reject. If it accepts, go to next step.

5 Otherwise, accept."

Intuitively, this proof works because if we are given a string w and it belongs to the concatenation of other languages, then there must be a particular way to split the string so that each part belongs to each language, and therefore to the concatenation of the languages. If there is no such way to split the string, then some part of it never belongs to one of the languages and therefore is not part of the concatenated language.

For Turing-recognizable languages, for any split of the string w there is some part of the string that does not belong in one of the languages. This means that its corresponding TM that recognizes that language will either reject or not halt (keep looping) on that part of the input string.

The only difference with the Turing-decidable case is that if the input string does not belong in the concatenated language, then there is a corresponding TM that decides one of the languages that will always reject (no looping).

1.2 Show that both the collection of Turing-decidable languages and the collection of Turing-recognizable languages are closed under intersection.

Intersection means that a string is both in language L_1 and L_2. In other words, their intersection is all strings that belong to both languages. Intuitively, we can form another proof by construction using this to guide us in the construction of the new TM.

Turing-decidable languages Let's say there are two languages L_1 and L_2 that are decided by two TMs T_1 and T_2. For a string w to be part of the intersection of L_1 and L_2, then it means both TMs T_1 and T_2 accept it. So let's construct a new TM T' that uses them as subroutines to recognize $L' = L_1 \frown L_2$.

T' = "On input w, where w is some string

1 Run T_1 on the input w. If T_1 rejects, reject. If T_1 accepts, go to the next step.

2 Run T_2 on the input w. If T_2 rejects, reject. If T_2 accepts, go to the next step.

3 Otherwise, accept.

Let's see how T' works. On step 1 it checks whether the input string is decided by T_1 (the decider for L_1). On the next step it does the same for T_2 and L_2. If the string is accepted by both TMs, then T' accepts it. This makes sense intuitively because that is the exact definition we used to describe the intersection of two languages.

Turing-recognizable languages

Again, the process is very similar. The construction is the same:

T' = "On input w, where w is some string

1 Run T_1 on the input w. If T_1 rejects, reject. If T_1 accepts, go to the next step.

2 Run T_2 on the input w. If T_2 rejects, reject. If T_2 accepts, go to the next step.

3 Otherwise, accept.

The only difference with decidable languages is that now, on the first and second steps T_1 and T_2 might just keep looping on the input string w and never halt. However, this is not a problem because it means they do not recognize the string and neither would T'.

1.3 Show that the collection of Turing-decidable languages are closed under complementation, but the collection of Turing-recognizable language is not.

Turing-decidable languages If a language L is Turing-decidable, it means that there is some TM T that decides it. The complement of the language L, is $\overline{L}$ - the set of all strings

that do not belong in L. So we can use this definition to construct a new TM T' that uses T as a subroutine to only accept strings outside of L:

T' = "On input w, where w is some string

1 Run T on the string w.

2 If T accepts, reject. If T rejects, accept.

Intuitively this makes sense because T' has the opposite steps of T so it will reject everything accepted by T, while also accepting everything rejected by T.

Turing-recognizable languages

What we did for the proof for decidable languages is that we essentially just swapped the accept and reject states of the TM. However, there is a reason why this would not work for recognizable languages. Let's say that the original TM kept looping on an input string w. This inherently means that w is never accepted by the TM and it was implicitly rejected. So when we construct the TM for the complement, this same input string w must be accepted by getting to the accept state in a finite amount of time. But how can it be accepted if the original TM never halts? This is the direction we can go in to formulate proof by contradiction.

From Theorem 4.11 in the textbook we know that:

Theorem 1 *A_{TM} is undecidable but recognizable.*

From Theorem 4.22 in the textbook we also know something else:

Theorem 2 *A language is decidable iff it is Turing-recognizable and co-Turing-recognizable.*

We can use these two theorems to show a contradiction and prove that the class of Turing-recognizable languages is not closed under the complement operation.

First, let's assume it is actually closed under complement.

We know that A_{TM} is recognizable. From our assumption it means that $\overline{A_{TM}}$ is also recognizable. If we follow 4.22, then A_{TM} would also be decidable since it is Turing and co-Turing recognizable. However, this leads to a contradiction with theorem 4.11 from which we know that A_{TM} is undecidable. Therefore our initial assumption must have been wrong. We can conclude that the class of Turing-recognizable languages is not closed under the complement operation.

2 Computability Theory

2.1 Determining Utility (#Computability, #Automata, #Logic)

A state q of an automaton M is *useless* if M does not enter q on any input.

2.1.1 Consider the problem of determining whether a pushdown automaton has any useless states. Formulate this problem as a language and show that it is decidable.

We need to frame this as a problem to construct a TM that decides this language. We can do it with the use of reduction.

Let's first define what the language is:

$$USELESS_{PDA} = \{\langle P \rangle | P \text{ has a useless state and } P \text{ is a PDA}\}$$

The general strategy for reduction to determine whether a language A is decidable is to find a language B to reduce A to $(A \rightarrow B)$. Since B is decidable, then so is A.

We can show that $USELESS_{PDA}$ is a decidable language by reducing it to E_{PDA}.

$$E_{PDA} = \{\langle P \rangle | P \text{ is a PDA and } L(P) = \varnothing\}$$

From theorem 4.8 from the textbook we are told that whatever is proved for CFGs it is also true for PDAs (since we can simply use a procedure to convert back and forth between the two). We know that E_{CFG} is decidable so we also know that:

Theorem 3 *E_{PDA} is decidable.*

Proof

Let's assume that some TM T_1 decides E_{PDA}. Now let's construct a TM T that decides $USELESS_{PDA}$.

$T =$ "On input $\langle P \rangle$, which is the string encoding of the PDA P

1 For each state q_i in the set of states of P repeat the following steps:

   2 Construct a new P' by using P and making q_i its only accept state.

   3 Run T_1 on $\langle P' \rangle$, where q_i is its only accept state.

   4 If T_1 accepts, accept.

5 Otherwise, reject.

Let's go over the logic behind this construction. If a state q of some PDA P is useless, then P does not enter q on any input. Therefore, if q was the only accept state of P, then the language P recognizes would be the empty language $L(P) = \varnothing$. So what the TM T does is it goes through all the states of P. Constructs a new PDA P' which is exactly the same but the only accept state is q_i. If that q_i is a useless state, then the PDA never enters it and cannot recognize any strings. Therefore its language is empty. The TM goes through all states. If this is the case for any of the states in P, then it means it has at least one useless state, so the TM T accepts P.

Another important thing to note is that in the formal proof by construction above, the input already belongs to both $USELESS_{PDA}$ to E_{PDA} since they take the same format (a PDA). So in step 2 the computable function f takes place to map the input from belonging to $USELESS_{PDA}$ to E_{PDA} but it still remains of the same type.

2.1.2 Show that the corresponding problem for Turing-machines is undecidable.

Proof Idea

This is a similar problem. However, here the general strategy for reducability to prove that a given language B is undecidable is to find another language A such that A is undecidable. The we need to reduce A to B to obtain a contradiction ($A \rightarrow B$).

Let's first define $USELESS_{TM} = \{\langle T, q\rangle\}|T$ is a TM and T has a useless state$\}$, where q is a useless state in T.

We can follow a similar approach to before and choose E_{TM} for our second language, where $E_{TM} = \{\langle M\rangle\}|M$ is a TM and $L(M) = \varnothing\}$. As per theorem 5.20 from the textbook we know that E_{TM} is undecidable.

Now, we will assume that $USELESS_{TM}$ is decidable, reduce E_{TM} to $USELESS_{TM}$ and show that leads to a contradiction.

Proof

Let's suppose that there exists a TM T that decides $USELESS_{TM}$. Let's use it to construct another TM M that decides E_{TM}:

$S =$ "On input $\langle M\rangle$, where M is a TM

1 Run T on $\langle M, q_{accept}\rangle$, where q_{accept} is the accept state of M.

2 If T accepts, then accept. Otherwise, reject.

Therefore, we proved that E_{TM} is decidable, which leads to a contradiction. This means that our initial assumption was wrong and there cannot exist a TM T that decides $USELESS_{TM}$. We can conclude that $USELESS_{TM}$ is undecidable.

A variable V in a context-free grammar G is *usable* if it appears in a derivation of some string $w \in L(G)$.

2.1.3 Given a CFG G and a variable V, consider the problem of testing whether V is usable. Formulate this problem as a language and show that it is decidable.

Let's first formulate this problem as a language:

$$USABLE_{CFG} = \{\langle G, V\rangle|G \text{ is a CFG and } V \text{ is a usable variable of } G\}$$

Proof Idea I will use a similar strategy to the proof of theorem 4.8 from the textbook, which says says that E_{CFG} is decidable.

The difference is that in this problem there are two pieces:

1) Is there a way to get from this variable V to an outputted string?

2) Is there a way to get from the start variable to this variable V?

To prove both parts our strategy remains the same as the strategy outlined in the proof of theorem 4.8. For this algorithm it does not matter if the symbols we initially mark are terminals or variables so we can treat the variable the same way.

Proof

1) Is there a way to get from this variable V to an outputted string?

So now to determine if some variable is usable it means that it needs to be used in the process of generating a string. Let's use the TM R constructed in the textbook to implement this: $R =$ "On input $\langle G, V \rangle$, where G is a CFG and V is one of its variables:

1 Mark all terminal symbols in G.

2 Repeat until no new variables get marked:

  3 Mark any variable A where G has a rule $A \to U_1 U_2 \dots U_k$ and each symbol $U_1, \dots, U_k$ has already been marked

3 If the variable V has not been marked, reject. Otherwise, go to the next step.

Now we still need to check for the second piece by extending this construction.

2) Is there a way to get from this variable V to an outputted string?

Currently we have a way to determine whether the variable V participates in the generation of some string. However, we need to make sure that the string $w \in L(G)$, meaning that there is a path from the start variable that goes through V to generate some string. So we need to be able to determine whether V is reachable from the start state.

4 Unmark all marked symbols. Mark the variable V.

5 Repeat until no new variables get marked:

  6 Mark any variable A where G has a rule $A \to U_1 U_2 \dots U_k$ and each symbol $U_1, \dots, U_k$ has already been marked

7 If the variable S has not been marked, reject. Otherwise accept.

We have finalized the formal proof by construction. Perhaps an improvement, just for the sake of clarity and separation of concerns, would be if we treat steps 1-3 as one TM M_1 and steps 4-7 as another TM M_2. Then our newly constructed TM R would use them as subroutines and will accept only if both M_1 and M_2 accept.

The reason it is decidable is that it will always halt. In the worst case scenario it just goes through all of the variables twice and then rejects or accepts based on the conditions. There is no step where it can start looping and not terminate. It follows the same logic as the proof for E_{CFG} in theorem 4.8 from Sipser.

A variable V in a context-free grammar G is *necessary* if V appears in every derivation of some string $w \in L(G)$.

2.1.4 Given a CFG G and a variable V, consider the problem of testing whether V is necessary. Formulate this problem as a language and show that it is T-recognizable, but undecidable.

$$NECCESSARY_{CFG} = \{\langle G, V\rangle | G \text{ is a CFG and } V \text{ is an unnecessary variable}\}$$

Recognizable

If the variable V is necessary, then when we remove it from the grammar G to create a new grammar G', their languages $L(G)$ and $L(G')$ would be different.

Proof Idea The general idea is to compare if all the strings that can be generated by G can also be generated by G'.

First we need to get the original grammar and variable of interest V. Remove V from G to create a new CFG G'. Get all the strings that can be produced by the grammar G. Check if G' can also generate all of them. If it cannot, then it means that the variable V was necessary for the production of some string. Following this conceptual idea, we can create a formal proof by construction.

To check whether G' can generate a particular string w we can use A_{CFG}, where

$$A_{CFG} = \{\langle G, w\rangle | G \text{ is a CFG that generates string } w\}$$

As per the Sipser textbook theorem 4.7, A_{CFG} is a decidable language. It can help us as a subroutine in the new TM we construct.

Proof

$R =$ "On input $\langle G, V\rangle$, where G is a CFG and V is a variable (not a terminal)

1 Get the CFL $L_G = L(G)$ as a list of all the strings that can be produced by G.

2 Remove the variable V from G to create another CFG G'.

3 Create a TM M that decided the language A_{CFG}.

4 For each string w in the list generated in step 1

  5 Run M on the input $\langle G', w\rangle$.

  6 If M rejects, accept. Otherwise, continue the loop.

7 Otherwise, reject.

We have described the algorithm that recognizes $NECCESSARY_{CFG}$. If $L(G')$ is different from $L(G)$, then R will accept at step 6 since the variable is necessary. But if the variable is unnecessary, then it will keep looping.

Undecidable I first approached this problem the wrong way. My initial idea was to use EQ_{CFG}, which is is an undecidable language as the textbook says, where

$$EQ_{CFG} = \{\langle G, H\rangle | G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$$

However, the strategy to prove that $NECCESSARY_{CFG}$ is undecidable would be to reduce EQ_{CFG} to $NECCESSARY_{CFG}$. The problem is that we cannot reduce EQ_{CFG} to $NECCESSARY_{CFG}$. If we do it the other way around, reduce $NECCESSARY_{CFG}$ to EQ_{CFG}, then $A \leq_m B$, and B is undecidable, then A could be either and we need more information.

Improved approach

I will use proof by contradiction where I reduce ALL_{CFG} to $NECCESSARY_{CFG}$.

$$ALL_{CFG} = \{\langle G\rangle | G \text{ is a CFG and } L(G) = \Sigma^*\}$$

Proof Idea

We are given some CFG G with a start variable S over an alphabet Σ and we want to determine whether $L(G) = \Sigma^*$. We can get G and transform it into G', where $L(G') = \Sigma^*$ for sure by adding a new variable in the following way:

1 Create a new start variable S' and add the following rule:

$$\text{S}' \ \rightarrow \text{S} \ \mid \ \text{N}$$

2 Add a rule for the new variable that can map onto any other symbol from the alphabet or the empty string. This guarantees that the grammar can generate any string in the language Σ^*

$$\text{N} \ \rightarrow \ \text{sN} \ \mid \ \epsilon \ \text{, for all s in } \Sigma$$

Let's put this in some machine M that returns the newly produced grammar G'.

In the way we have constructed G', it is able to generate all strings that G can and possibly, if $L(G) \neq \Sigma^*$), then G' can generate some additional strings.

So we need to check if G actually needed this new variable N to begin with.

Case 1: If $L(G) \neq \Sigma^*$, then this new variable N is necessary to produce all strings that G' can generate. In that case $\langle G', N\rangle \in NECESSARY_{CFG}$.

Case 2: If $L(G) = \Sigma^*$, then the new variable N is not necessary to produce all the same strings. In that case $\langle G', N\rangle \notin NECESSARY_{CFG}$.

So $G \in ALL_{CFG}$ iff $\langle G', N\rangle \notin NECESSARY_{CFG}$.

Proof

Assume that there is a TM decider T for $NECESSARY_{CFG}$.

$R =$ "On input G, where G is a CFG

1 Run machine M on input G to get G' (create G' using the description above).

2 Run T on input $\langle G', N \rangle$.

3 If T accepts, reject. If T rejects, accept.

This leads to a contradiction. If the language $NECESSARY_{CFG}$ is decidable, then we proved that the language ALL_{CFG} is decidable too. However, from Sipser we already know that ALL_{CFG} is in fact undecidable. This is where the contradiction lies. Therefore, our initial assumption was wrong and $NECESSARY_{CFG}$ cannot be a decidable language.

2.2 Hello World!(#Computability, #Logic, #Machines)

An enumerator E is called a *Hello-World-program*, if the following is true: When given the empty string ε as input, E prints the string 'Hello World'. Consider the language

$$HW = \{\langle E \rangle | E \text{ is a Hello-World-program}\}.$$

Your task is to show HW is undecidable.

For each enumerator E and a string w, define another enumerator $P_{\langle E,w \rangle}$ as follows:

$P_{\langle E,w \rangle}$ = "On input $\langle x \rangle$ where x is a string:

1 **Simulate E on w.**

2 **Print 'Hello World'.**"

2.2.1 Show that E halts on w iff $P_{\langle E,w \rangle} \in HW$.

Since we are interested in whether a language $\in HW$, then it means that we need to feed the empty string into $P_{\langle E,w \rangle}$ by the definition of HW languages. So looking at the steps above x is replaced with ϵ.

The expression iff describes a 2-way relationship: either both sides are true or both sides are false.

Both are true If $P_{\langle E,w \rangle} \in HW$, then we know that it is an enumerator which prints 'Hello World' when given the empty string ϵ as input. From $P_{\langle E,w \rangle}$'s description above, we can see that printing the statement is the second step. To get to it, it means that the first step (Simulate E on w) was executed and also terminated at some point (halted). Therefore, when E was simulated on w, it halted which allowed $P_{\langle E,w \rangle}$ to move to step 2 and print 'Hello World', making it $\in HW$.

Both are false If $P_{\langle E,w \rangle} \notin HW$, then we know that E never halted in step 1. The process is that if E never halts, then on input string ϵ, $P_{\langle E,w \rangle}$ would never get past simulating E on w in step 1. Therefore, it would never print 'Hello World' so $P_{\langle E,w \rangle} \notin HW$.

Suppose to the contrary that there exists a TM R that decides HW. Define another TM S that uses R as follows:

S = "On input $\langle E, w\rangle$ where E is an enumerator and w is a string:

1 **Simulate R on $P_{\langle E,w\rangle}$.**

2 **If R accepts, accept.**

3 **If R rejects, reject.**"

Complete the proof by showing each of the following statements hold.

2.2.2 S halts on every input $\langle E, w\rangle$.

Proof Idea The main idea behind deducing this is that we just assume that R is a decider. When you simulate R on anything then you will get an answer.

Proof

We assume that there exists a TM R that is a decider, meaning that it halts (accepts or rejects) on any input string $\langle E, w\rangle$. Since S uses R as a subroutine and it simply accepts if R accepts or rejects if R rejects (and R is guaranteed to do one or the other), that means that S will also always halt.

Proof using formal logic

We can also express this in terms of propositional logic using the inference rule of Modus Ponens. First, let's define the atomic sentences:

- $HALT_R$ = The TM R halts.
- $HALT_S$ = The TM S halts.

Given the TM as a system the logic is that if R halts, then S also halts: $HALT_R \implies HALT_S$. Now we can construct the full argument using Modus Ponens:

$$\therefore \frac{HALT_R \implies HALT_S, HALT_R}{HALT_S}$$

The notation means that when a sentence of the form if-then: "If the TM R halts, then S halts." is given together with the left-hand side as a given ($HALT_R$, since we assume that R always halts), then the conclusion on the right-hand side can be inferred ($HALT_S$, S halts).

2.2.3 If E halts on input w, then S accepts $\langle E, w\rangle$.

Proof

Let's trace back the steps of the process one by one. First, we give S some input $\langle E, w\rangle$. Then in step 1, S simulates R on $P_{\langle E,w\rangle}$ where the input is the empty string, ϵ. Now we go to $P_{\langle E,w\rangle}$'s steps. As a reminder this is what it does:

$P_{\langle E,w\rangle}$ = "On input $\langle x\rangle$ where x is a string:

1 **Simulate E on w.**

2 **Print 'Hello World'.**"

In step 1 we simulate E on w. We are given that E halts, so we go to step 2, where the string 'Hello World' is printed. This means that $P_{\langle E,w\rangle} \in HW$. Therefore, R accepts.

Now we go back to the steps in S. We have completed step 1. We go to step 2 where S accepts since R accepted.

Proof with formal logic

We can also get to this conclusion using propositional logic. Let's define the atomic sentences:

- $ACCEPT_R$: The TM R accepts (language $\in HW$).
- $REJECT_R$: The TM R rejects (language $\notin HW$).
- $HALT_E$: The enumerator E halts on w.
- $L(P_{\langle E,w\rangle})_{HW}$: The language of $P_{\langle E,w\rangle} \in HW$.

If E halts, then $L(P_{\langle E,w\rangle}) \in HW$.

- $HALT_E \implies L(P_{\langle E,w\rangle})_H W$

If $L(P_{\langle E,w\rangle}) \in HW$, then R will accept it.

- $L(P_{\langle E,w\rangle})_{HW} \implies ACCEPT_R$

If R accepts, then S accepts.

- $ACCEPT_R \implies ACCEPT_S$

Now through a process of forward chaining by repeatedly applying Modus Ponens together with the help of our rules which are in the Knowledge base above, we can show how the conclusion that S halts is derived.

Our first premise is that E halts on input w.

$$\begin{array}{l} HALT_E \\ HALT_E \implies L(P_{\langle E,w\rangle})_{HW} \\ \hline \therefore \quad L(P_{\langle E,w\rangle})_{HW} \end{array}$$

Now the conclusion is used as a premise in the next application of Modus Ponens:

$$\begin{array}{l} L(P_{\langle E,w\rangle})_{HW} \\ L(P_{\langle E,w\rangle})_{HW} \implies ACCEPT_R \\ \hline \therefore \quad ACCEPT_R \end{array}$$

We repeat the process again:

$$\begin{array}{l} ACCEPT_R \\ ACCEPT_R \implies ACCEPT_S \\ \hline \therefore \quad ACCEPT_S \end{array}$$

Using formal logic, we showed how the conclusion that S accepts necessarily follows from the premise that E halts and the additional premises in the form of rules. This is a valid deductive argument since the conclusion follows from the premises and it is sound since the premises are all true.

2.2.4 If E loops on input w, then S rejects $\langle E, w \rangle$.

Proof

To deduce this we can follow a similar logic as before with one minor difference.

Let's trace back the steps of the process one by one. First, we give S some input $\langle E, w \rangle$. Then in step 1 S simulates R on $P_{\langle E,w \rangle}$ where the input is the empty string, ϵ. Now we go to $P_{\langle E,w \rangle}$'s steps.

In step 1 we simulate E on w. We are given that E loops, so $P_{\langle E,w \rangle}$ never goes to step 2 and the string 'Hello World' is printed. This means that $P_{\langle E,w \rangle} \notin HW$. Therefore, R rejects.

Now we go back to the steps in S. We have completed step 1. We go to step 2 where S does not do anything since R did not accept. Then S goes to step 3 where it rejects since R rejected.

In the next sub-question we will see that this is exactly where the contradiction comes from. There is a problem with the fact that if E loops, then S rejects. There is a reason why this is a problem. If we have something that would decide even though we have an enumerator that is looping - it gives us a way to solve another undecidable problem (this will be explored in the next sub-problem).

Proof with formal logic

Let's define some new atomic sentences that we need.

- $ACCEPT_R$: The TM R accepts (language $\in HW$).
- $REJECT_R$: The TM R rejects (language $\notin HW$).
- $HALT_E$: The enumerator E halts on w.
- $L(P_{\langle E,w \rangle})_{HW}$: The language of $P_{\langle E,w \rangle} \in HW$.

Let's define some new rules to add to our Knowledge base that we need for the proof.

If E loops, then $P_{\langle E,w \rangle} \notin HW$.

- $\neg HALT_E \implies \neg L(P_{\langle E,w \rangle})_{HW}$

If $L(P_{\langle E,w \rangle}) \notin HW$, then R will reject it.

- $\neg L(P_{\langle E,w \rangle})_{HW} \implies REJECT_R$

If R rejects, then S rejects.

- $REJECT_R \implies REJECT_S$

Now we go through the same process of forward chaining by repeatedly applying Modus Ponens as we did before.

Our first premise is that E loops on input w.

$$\begin{array}{l} \neg HALT_E \\ \neg HALT_E \implies \neg L(P_{\langle E,w\rangle})_{HW} \\ \hline \therefore \quad \neg L(P_{\langle E,w\rangle})_{HW} \end{array}$$

Now the conclusion is used as a premise in the next application of Modus Ponens:

$$\begin{array}{l} \neg L(P_{\langle E,w\rangle})_{HW} \\ \neg L(P_{\langle E,w\rangle})_{HW} \implies REJECT_R \\ \hline \therefore \quad REJECT_R \end{array}$$

We repeat the process again:

$$\begin{array}{l} REJECT_R \\ REJECT_R \implies REJECT_S \\ \hline \therefore \quad REJECT_S \end{array}$$

Using formal logic, we showed how the conclusion that S rejects necessarily follows from the premise that E loops and the additional premises in the form of rules that we added to our knowledge base.

2.2.5 The language HW is undecidable.

There are two ways to prove this.

The first way is to use reduction, where the general strategy is the following: if we are trying to show that language B is undecidable, then find an appropriate undecidable language A, show that there is a reduction that maps from A to B. Since A is undecidable, B must also be undecidable.

Reduction $HALT_{TM} \leq_m HW$

Since the problem has building up on the second approach with reduction, this is the approach I will use. Let's take a close look at the TM S that we constructed. It is a machine that decides the language $HALT_{TM}$

$$HALT_{TM} = \{\langle M, w\rangle | M \text{ is a TM and } M \text{ halts on input } w\}$$

In S the input is E and w. Step 1 is where the function converts the input from input in $HALT_{TM}$ to input in HW. S uses R as a subroutine. If R accepts, then it means the string 'Hello World' was printed. Therefore, we can infer that E halted on w. In that case S also accepts.

In the other case, where R rejects, it means the string 'Hello World' was not printed. Therefore, we can infer that E did not halt on w. In that case S also rejects.

What we have seen is that we constructed a decider for the language $HALT_{TM}$ and it is therefore a decidable language. However, from the Sipser textbook we know that $HALT_{TM}$ is, in fact, undecidable. This is a contradiction.

We can conclude that our initial assumption was wrong and that the language HW is undecidable.

Reduction $A_{TM} \leq_m HW$

Another possible language we can use for the reduction is A_TM. I will construct this proof just for fun. We already know that A_TM is reducible to the halting problem from the textbook. Reference theorem 5.1 in Sipser (the proof is very similar).

$$A_{TM} = \{\langle M, w\rangle | M \text{ is a TM and } M \text{ accepts } w\}$$

Let's construct a TM K that decides A_{TM}:

K = "On input $\langle E, w\rangle$, where E is an enumerator and w is a string:

1 Simulate R on $P_{\langle E,w\rangle}$.

2 If R rejects, reject.

3 If R accepts, simulate E on w until it halts.

4 If E has accepted, accept; if E has rejected, reject.

If R decides HW, then K decides A_{TM}. Because A_{TM} is undecidable, then HW is also undecidable.

Proof using co-Turing recognizable

We can use Theorem 4.22 from Sipser to prove that HW is undecidable.

Theorem 4 *A language is decidable iff it is Turing-recognizable and co-Turing-recognizable.*

Let's construct a recognizer for HW:

R_1 = "On input $\langle E, w\rangle$, where E is an enumerator and w is a string:

1 Simulate $P_{\langle E,w\rangle}$ on the empty string.

2 If $P_{\langle E,w\rangle}$ ever prints 'Hello World', accept. If $P_{\langle E,w\rangle}$ never prints 'Hello World', reject.

This machine loops on input $\langle E, w\rangle$ if E keeps looping on w, which is why it is not a decider.

Now we can attempt to construct a recognizer for $\overline{HW}$.

R_2 = "On input $\langle E, w\rangle$, where E is an enumerator and w is a string:

1 Simulate $P_{\langle E,w\rangle}$ on the empty string.

2 If $P_{\langle E,w\rangle}$ ever prints 'Hello World', reject. If $P_{\langle E,w\rangle}$ never prints 'Hello World', accept.

The reason this machine is not a recognizer is that if $\langle E, w\rangle$ if E keeps looping on w, then it will also keep looping on input $\langle E, w\rangle$. It will never accept and therefore, never recognize some of the inputs.

Therefore, $\overline{HW}$ is not Turing-recognizable. According to theorem 4.22, this means that HW is not decidable.

3 Reflection

3.1 What computational concepts from this unit are you curious to know more about? Give one example of a question about the material that you'd like to explore further, and describe why this is an interesting question to you.

Question: How is computability theory related to Computer Science and how did it help advance?

This question is interesting to me because I want to understand the role this theory placed in the development of Computer Science and also connect it to a practical problem and a field I am interested in.

I would also like to explore how are/were PDAs used. They seem like a big improvement from FAs so it would be interesting to learn more.

3.2 Give two examples HCs you applied while solving this assignment. How did you apply the HC? Was the application successful? If yes, why? If not, how could you improve your application?

#networks

This application was not standard and a bit unexpected. I applied it in problem 2.1 where we had to determine whether a state q of an automaton M is useless. To construct the proofs for both 2.1.1 and 2.2.2 I build the idea on the structure of the states and how they are connected with each other. For example, in 2.1.1 it was based on the structure of a pushdown automaton. I analyze the network and use the fact that if only one state accepts and the language is empty, then it means the PDA never entered that state. So I use this effective intervention to construct a new PDA and check if a useless state exists in the system. Usually when we talk about #networks we reference aspects like connectivity, dispersity, etc. However, the beauty of the proof here lies in the fact that it is generalizable. No matter how many nodes there are, how they are connected (how the transition function is defined), it will always be able to determine if there is a useless state.

#strategize

When the assignment was first published, I went through all of it and read it thoroughly. Then, I referenced my grades from class and from the first assignment to identify my weaknesses and strengths. I saw that my biggest weakness is the LO #logic and that every problem from the three on the assignment targets that specific LO. On the bright side, it was an opportunity for me to practice and improve by using the feedback from before. I also identified that I am strong in the other targeted LOs and I know how to apply them well based on the previous scores I have received. Then, I developed a strategy on how to approach the assignment. First, I went though my previous feedback and brainstormed all aspect of #logic that are unclear to me. Then, I asked some classmates how they applied the LO successfully in our previous assignment. When building the arguments, I would often break them into atomic sentences and build them using formal logic. This helped me make them more concrete and clear throughout the whole assignment. Overall, it was a good application that allowed me to successfully complete the assignment before the deadline.