CSE 105: Introduction to the Theory of Computation Fall 2015

Lecture Notes: FST and Mapping Reductions

Instructor: *Daniele Micciancio* UCSD CSE

Earlier in the course we have proved that given a DFA M and an FST T, one can build a DFA M' computing their function composition $M'(w) = M(T(w))$. What's the relation between the languages $L = \mathcal{L}(M)$ and $L' = \mathcal{L}(M')$ of the two automata? It follows by construction that L' is precisely the preimage of L under the function $f(w) = FST(w)$ computed by the transducer:

$$L' = f^{-1}(L) = \{w \mid f(w) \in L\}.$$

This is naturally formualted as a closure property of regular languages: for any FST-computable function $f\colon \Sigma^* \to \Gamma^*$, if $L \subseteq \Gamma^*$ is regular, then also $L' = f^{-1}(L) \subseteq \Sigma^*$ is regular.

Notice that the relation $L' = f^{-1}(L)$ can be equivalently expressed as the condition

$$\forall w. w \in L' \iff f(w) \in L.$$

As usual, this double implication can be rewritten as two separate properties:

1. $w \in L' \Rightarrow f(w) \in L$, and
2. $w \notin L' \Rightarrow f(w) \notin L$.

A function $f\colon \Sigma^* \to \Gamma^*$ satisfying these properties is called a *reduction* from L' to L. The name *reduction* comes from the fact that you can think of f as a method to translate (membership) questions about L' to (membership) questions about L. So, the task of determining if $w \in L'$ is reduced to the task of determining if $f(w) \in L$. For simplicity of exposition, below, we assume all languages are over some fixed alphabet Σ.

Definition 1 *For any two languages $A, B \subseteq \Sigma^*$, a* reduction *from A to B is a function $f\colon \Sigma^* \to \Sigma^*$ such that for all $w \in \Sigma^*$,*

- *if $w \in A$ then $f(w) \in B$,and*
- *if $w \notin A$ then $f(w) \notin B$.*

A reduction f is FST-computable *if it is computed by a Finite State Transducer T. We say that A is* FST-reducible *to B (in symbols $A \leq_{FST} B$) if there is an FST-computable reduction f from A to B.*

Reductions are one of the most important concepts in the study of the theory of computation, and one can define many different flavors of reductions. But for now, we will keep the discussion focused on regular languages. The notation $A \leq B$ (read "A reduces to B") can be interpreted as a comparison between the "hardness" of the two problems. Think of problems that can be solved by a DFA (i.e., regular languages) as being "computationally easy", and problems that cannot be solved by a DFA (i.e., nonregular languages) as being "compuptationally hard". The closure property showing that if L is regular, then $f^{-1}(L)$ is also regular (for any FST-computable function f) can be reformulated as follows:

Corollary 1 *If $A \leq_{FST} B$ and B is regular, then A is also regular.*

Informally, if A is not harder than B and B is "easy", then you can conclude that A is also "easy", where "easy" means that the membership decision problem can be solved by a DFA. By taking the contrapositive, we also get that if A is not harder than B and A is hard, then you can conclude that also B must be hard.

Corollary 2 *If $A \leq_{FST} B$ and A is not regular, then B is also not regular.*

Reductions are a powerful tool to study the complexity of computational problems, and can be used both to prove that certain problems are computationally easy, and others are computationally hard. But be careful: you need to use reductions in the correct direction. For example, if you show that $A \leq_{FST} B$ and also prove (or know) that A is regular, you cannot draw any conclusion about B: the problem B may be regular or nonregular. Similarly, if $A \leq_{FST} B$ and B is nonregular, you cannot conclude anything about the regularity of A.

Reductions with Turing machines Reductions are particularly useful in studying Turing-recognizability and decidability of problems. In the textbook, the reductions used in this context are called *mapping reductions*, and denoted $A \leq_m B$, and you can read about them at length in chapter 5 of the textbook.[1] The definition is the same as before: f is a reduction from A to B if for every $x \in \Sigma^*$, $x \in A$ if and only if $f(x) \in B$. The difference with the FST-reductions defined above is that the function f describing a mapping reduction is not necessarily computable by an FST: f can be any algorithmically computable function, i.e., a function f which can be computed by a Turing machine. Technically, you can think of a TM computing a function $f(x)$ by starting with the input x written on its tape, and terminating with the result $f(x)$ written on the tape. But typically the TM model is only used informally when describing reductions, so this level of detail is not required.

Now, let A and B be two languages, and assume $A \leq_m B$. Remember, this means that there is a function f such that $A = f^{-1}(B)$ and f is Turing computable, i.e., there is a

[1]The textbook uses the term "mapping reductions" to distinguish them from still another type of reductions, called "Turing reductions", which are introduced first. Turing reductions correspond to using arbitrary closure properties to prove undecidable languages: you start by assuming there exists a Turing machine deciding a language B, and then use it to build a Turing machine to decide a different language A. By conrapositive, it follows that if A is undecidable, then also B is undecidable.

Turing machine T such that for any input $w \in \Sigma^*$, running $T(w)$ outputs $f(w)$. Clearly, if $A \leq_m B$ (i.e., A is map-reducible to B) and B is regular, you cannot conclude that A is regular: combining a DFA M for B with the Turing machine T computing the function f, will not give you a DFA as a result: you can still combine M and T to get a machine M' that computes their composition $M'(w) = M(T(w))$, but in general M' will be another Turing machine. In fact, using the power of Turing machines, it is very easy to build such an M' by running T and M sequentially: on input w, the new machine $M'(w)$

1. first runs T on input w, to compute the intermediate result $u = f(w)$, and then

2. runs M on u to come up with the final acceptance/rejection decision.

So, A will not be regular, but it is certainly decidable, because it is the language of the Turing machine $M'(w) = M(T(w))$, and this Turing machine is a decider, i.e., it terminates on all inputs w. Notice that you don't need to start from a DFA M. Even if you combine T with an arbitrary Turing machine M with $\mathcal{L}(M) = B$ as described above, you will still get a Turing machine M' such that $\mathcal{L}(M') = f^{-1}(B) = A$. This proves that if $A \leq_m B$ and B is Turing recognizable, then A is also Turing recognizable. Moreover, if M is a decider, i.e., it always terminates, then also M' will also terminate on all inputs. So, we also have that if $A \leq_m B$ and B is decidable, then A is also decidable.

Theorem 1 *If $A \leq_m B$ and B is Turing recognizable, then A is also Turing recognizable. Moreover, if B is decidable, then A is also decidable.*

Taking the contrapositive, we also get the following corollary that can be used to prove that languages are undecidable, or non-recognizable by Turing machines that are not required to terminate on every input.

Corollary 3 *If $A \leq_m B$ and A undecidable, then B is also undecidable. Moreover, if A is not even Turing recognizable, then B is also not Turing recognizable.*

As before, you may think of reductions as a way to compare the computational hardness of two languages: if A is "computationally hard" and $A \leq_m B$, then B is also "computationally hard", where "computationally hard" may mean undecidable, or unrecognizable.

Undecidable problems So, are there problems that cannot be solved by Turing machines? Of course there are, as we can build one by diagonalization. To this end, we need a way to encode Turing machines as string, similarly to what we did for regular expressions, context free grammars, and other automata. Since the specific details of the encoding are irrelevant, let us just assume some fixed encoding function $\langle M \rangle$ that can be used to represent Turing machines M as strings over some alphabet Σ.

Consider the following "diagonal language":

$$D = \{\langle M \rangle \colon \langle M \rangle \notin \mathcal{L}(M)\}.$$

Clearly, for any Turing machine M, $D \neq \mathcal{L}(M)$.[2] In other words, D is not the language of any Turing machine, i.e., D is not Turing recognizable.

This, by itself, may not seem very interesting, because the problem described by D is of unclear practial relevance: why would you even want to know if a computer program acceps its own description? But here is where reductions come into play: we can use mapping reductions to show that many other problems (potentially relevant to computer practice) are at least as hard as D, and therefore they are not algorithmically solvable.

Consider the halting problem: given a program (represented by a Turing machine M) and an input w, determine if the computation $M(w)$ terminates or runs forever. (Notice, in this specific problem we don't care about the accept/reject result of the computation. We are only interested in detecting infinite loops.) In order to compare this problem to D we need first for formally define it as a decision problem, or language. Define

$$Halt = \{\langle M, w\rangle : M(w) \text{ terminates.}\}$$

This problem is Turing recognizable: in order to termine if $\langle M, w\rangle \in Halt$, you can simply run M on input w, and accept if the computation $M(w)$ terminates. If the computation does not terminate, we will go into an infinite loop. So, this only shows that $Halt$ is Turing recognizable. The question is: is $Halt$ decidable? We will show that it is not by giving a reduction from D to the complement of $Halt$. This will prove that the complement of Halt is undecidable. But since decidable languages are closed under complementation, we can conclude that $Halt$ is also undecidable.

Here is how the mapping reduction works. Recall, we want to reduce D to $\overline{Halt}$. So, we start from an instance $w = \langle M\rangle$ of the diagonal language D, and map it to an instance $\langle M', w'\rangle$ of $\overline{Halt}$ as follows. The Turing machine $M'(x)$ is built from $w = \langle M\rangle$ as follows:

1. if $w = \langle M\rangle$ is syntactically malformed (e.g., if M is not even a Turing machine), then $M'(x)$ immediately accepts every string x.

2. Otherwise, $M'(x)$ runs the Turing machine M on its own encoding w. If $M(w)$ accepts, then $M'(x)$ also accepts. Otherwise, $M'(x)$ enters an infinite loop.

Notice that, in the above construction, the Turing machine $M'(x)$ ignore its input.[3] So, the choice of the input x is not important, and we may fix $x = \epsilon$. The output of the reduction is

$$f(\langle M\rangle) = \langle M', \epsilon\rangle.$$

This function is Turing computable: one can algorithmically check if w is a syntactically valid description of a Turing machine, and produce the code of another Turing machine M'

[2]If you don't see the proof yet, you should go back and read the notes on diagonalization for regular languages. Then, on your own, build a language that is not context free using the same diagonalization method applied to context free grammars, and prove that your answer is correct.

[3]In particular, $\mathcal{L}(M)$ will be either $\emptyset$ or Σ^*. But this is not a critical feature of the reduction. We are pointing this out just for clarification and avoid possible confusion: The Turing machine M' built by the reduction is not solving any particularly challenging problem.

that behaves as described. Notice: the reduction takes w as input, and it outputs a string $\langle M', \epsilon \rangle$ that contains a description of M'. The reduction only needs to print out the code of M', and it never actually runs M' on any input. So, the reduction terminates in finite time, whether or not $M'(\epsilon)$ gets into an infinite loop.

In order to show that the reduction is correct we neet to prove that

- if $w \in D$ then $f(w) \in \overline{Halt}$,
- if $w \notin D$ then $f(w) \notin \overline{Halt}$.

It remains to verify these two properties. First assume $w \in D$, i.e., $w = \langle M \rangle$ for some Turing machine M such that $w \notin \mathcal{M}$. It follows that $M'(\epsilon)$ does not terminate, either because $M(w)$ goes into an infinite loop, or because $M(w)$ rejects. So, $f(w) = \langle M', \epsilon \rangle \notin Halt$ as required.

Next, consider the case when $w \notin D$. This time, either w is not a valid Turing machine, or $w = \langle M \rangle$ for a Turing machine M such that $M(w)$ accepts. In both cases, the Turing machine $M'(x)$ built by the reduction is a machine that accepts every input x. In particular, $M'(\epsilon)$ terminates, and $f(w) = \langle M', \epsilon \rangle \in Halt$. Therefore $f(w) \notin \overline{Halt}$, completing the proof that f is a reduction from D to $\overline{Halt}$.