

## Reducibility and Unsolvability

A new technique surfaced while proving that the problem of whether or not a Turing machine halts for all inputs is unsolvable. This technique is called *reducibility*. It has traditionally been a very powerful and widely used tool in theoretical computer science as well as mathematics. We shall benefit greatly by investigating it further.

What happened in that proof was to transform an old problem into a new problem by taking instances of the first problem and translating them into instances of the new problem.

We did this also in the proof of the halting problem. Let us closely examine exactly what happened from the point of view of set membership.

First, recall the definition of the diagonal set  $K = \{ i \mid M_i(i) \text{ halts} \}$  and define the set of pairs of integers for the general halting problem as  $H = \{ \langle i, x \rangle \mid M_i(x) \text{ halts} \}$ . We noted that:

$$i \in K \text{ meant exactly the same as } \langle i, i \rangle \in H.$$

We then claimed that we could construct a decision procedure for membership in  $K$  based upon deciding membership in  $H$ . In other words, to solve membership in  $K$ , just take a candidate for membership in  $K$  and change it into a candidate for membership in  $H$ . If this new element is in  $H$ , then the original one had to be in  $K$ .

So, we transformed the membership problem for  $K$  into that for  $H$  by mapping or translating one integer into a pair as follows.

$$i \rightarrow \langle i, i \rangle.$$

(Then, of course, we noted that since membership in  $K$  is unsolvable there is no way that membership in  $H$  could be solvable since that would imply the solvability of membership in  $K$ .)

Now let's return to the proof that deciding whether or not a Turing machine accepts all inputs is unsolvable. Again, we translated a known unsolvable problem into the problem whose unsolvability we were trying to prove. In particular, we took an arbitrary machine  $M_i$  and constructed another Turing machine  $M_{\text{all}}$  such that if  $M_i$  halted on its own index (the integer  $i$ ), then  $M_{\text{all}}$  halted for all inputs. Then we claimed (and proved) that if the new problem

(halting for all inputs) were solvable then the old one (a Turing machine halting on its own index) had to be solvable too. Thus the new problem must have been unsolvable as well.

This tells us something about the concept of unsolvability as well as providing a new useful technique. Mathematicians often believe that properties that are preserved by mappings or transformations are the most important and in some sense the strongest properties that exist. Thus unsolvability must be a rather serious concept if we cannot get rid of it by changing one problem into another!

OK, back to mappings. Not only did we transform one problem into another, but *we did it in a computable or effective manner*. We took an arbitrary machine  $M_i$  and noted that we would like to ask later if  $M_i(i)$  halts. From  $M_i$  we built another machine  $M(x, i)$  which was merely  $M_u(i, i)$ . We then trotted out Church's thesis and claimed that there was an integer  $a$  such that we designed was the machine  $M_a(x, i)$ . At this point we invoked the s-m-n theorem and asserted that there was a computable function  $s(a, i)$  such that

$$M_{s(a, i)}(x) = M_a(x, i) = M(x, i) = M_u(i, i) = M_i(i).$$

And in this manner we were able to show that:

$$M_i(i) \text{ halting means exactly the same as } M_{s(a, i)}(x) \text{ halting for all } x$$

Thus we used the function  $s(a,i)$  to map between problems. Now we turn our attention back to sets since we often formulate things in terms of sets. In fact, sets can be built from problems. Recall that the machine  $M_i$  accepts the set  $W_i$  (or  $W_i = \{ x \mid M_i(x) \text{ halts} \}$  ) and consider the set definitions in the chart below.

<b>K</b>	<b>{ i   <math>M_i(i)</math> halts }</b>	<b>{ i   <math>i \in W_i</math> }</b>
<b>E</b>	<b>{ i   <math>M_i(x)</math> halts for all x }</b>	<b>{ i   <math>\forall x [x \in W_i]</math> }</b>

Note that by showing ' $M_i(i)$  halts if and only if  $M_{s(a, i)}(x)$  halts for all  $x$ ' we were also demonstrating that:

$$i \in K \text{ if and only if } s(a, i) \in E$$

thus transforming the membership problem for one set (K) into the membership problem for another set (E).

One additional housekeeping item needs discussing. The Turing machine index  $a$  is actually a constant rather than a variable since it came from the definition of the particular machine  $M(x, i) = M_u(i, i)$ . Thus it does not need to be a parameter since it never varies.

With this in mind, we define the function  $g(i)$  to be  $s(a, i)$ . Thus  $M_{g(i)}(x)$  is exactly the same machine as  $M_{s(a, i)}(x)$  and:

$$i \in K \text{ if and only if } g(i) \in E.$$

Summing up, what we have done is translate (or map) the set  $K$  into the set  $E$  using the computable function  $g(i)$ . The Venn diagrams in figure 1 illustrate this. Note that the function  $g()$  maps all of  $K$  into a portion of  $E$  and all of the elements not in  $K$  into a portion of  $E$ 's complement.

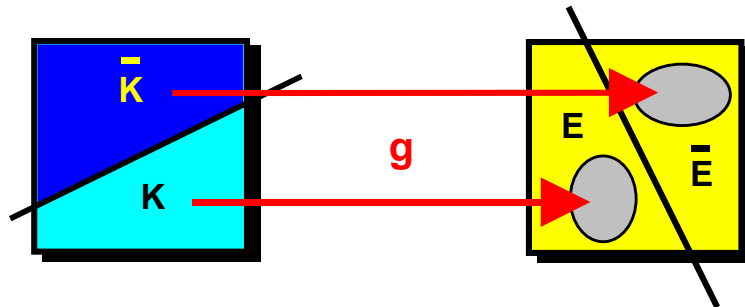


Figure 1 - Mapping between  $K$  and  $E$

Note that the mapping does not map  $K$  into *all* of  $E$ . When given Turing machines whose indices are members of  $K$ , it transforms them into new Turing machines that accept all inputs and thus have indices that are members of  $E$ .

And, the mapping produces members of  $E$  that operate in a very special manner. They make up a family of machines that simulate the original machine's execution with its own index as input  $[M_i(i)]$ . Since there are many machines that do not perform computations like this, yet halt on all inputs, it is clear that only a portion of  $E$  is mapped into.

The formal definition of reducibility between sets appears below. In general it is exactly the same as the mappings we have described in the discussion above and illustrated in figure 1.

**Definition.** *The set  $A$  is **reducible** to the set  $B$  (written  $A \leq B$ ) if and only if there is a totally computable function  $g$  such that for all  $x$ :*

$$x \in A \text{ if and only if } g(x) \in B.$$

That is what we used to do the construction in the last theorem of the last section. It was just a transformation from the set  $K$  to the set  $E$ . Now we shall tie this into solvability and unsolvability with our next theorem.

**Theorem 1.** *If A is reducible to B and membership in B is solvable, then membership in A is solvable also.*

**Proof.** This is very straightforward. First, let  $A \leq B$  via the function  $g$ . (In other words,  $\forall x[x \in A \text{ iff } g(x) \in B]$ .) Now, suppose that the Turing machine  $M_b$  solves membership in the set B. Recall that this means that  $M_b$  outputs a one when given members of B as input and outputs zeros otherwise.

Consider the machine built from  $M_b$  by preprocessing the input with the function  $g$ . This gives us the machine  $M_b(g(x))$  which we shall claim solves membership in A since  $g$  maps members of A into members of B. That is, for all  $x$ :

$$\begin{aligned} x \in A & \text{ iff } g(x) \in B && \text{[since } A \leq B\text{]} \\ & \text{ iff } M_b(g(x)) = 1 && \text{[since } M_b \text{ solves B]} \end{aligned}$$

If  $M_b(g(x))$  is an actual Turing machine, we are done. We claim it is because  $M_b$  is a Turing machine and the function  $g$  is computable (thus there is some  $M_g$  that computes it). Church's thesis and the s-m-n theorem allow us to find the index of this machine that solves membership in the set A in our standard enumeration.

This leads to an immediate result concerning unsolvability.

**Corollary.** *If A is reducible to B and membership in A is unsolvable then so is membership in B.*

Let's pause and discuss what we have just discovered. From the theorem and corollary that we just looked at, we can see that we are able perform only the mappings illustrated in figure 2.

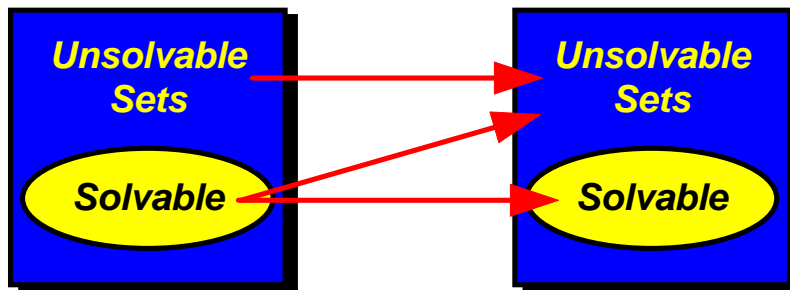


Figure 2 - Allowable reducibility mappings

Note that if a set has a solvable membership problem then we can map from it to any other set except the empty set (since there is nothing there to map into), or the set of all integers (which has an empty complement). But, if a set has an

unsolvable membership problem, we are only able to map from it to unsolvable sets. On the other hand, we are only able to map into solvable sets from other solvable sets. Here is a summation of this in the chart below.

*Assume that A is reducible to B*

<i>Then if:</i>	<i>We know:</i>
A is unsolvable	B is unsolvable
A is solvable	Nothing about B
B is solvable	A is solvable
B is unsolvable	Nothing about A

There is some insight about computation to be gained from this. If a set has an unsolvable membership problem then it must be rather difficult to decide whether an integer is a member of the set or not. (That of course is a droll and obvious understatement!) It is certainly more difficult to decide membership in an unsolvable set than one with a solvable membership problem since we are able to do the later but not the first. Thus reducibility takes us from easier problems to more difficult ones. Reducibility (and the intuition of mapping from easy to hard problems) shall surface again in the study of the complexity of computation where we are concerned with the time and space necessary to solve problems and perform computation.

Reducibility provides an additional method for determining the solvability of membership in a set. Here is a summation:

***To show that a set has a solvable membership problem:***

- ***construct a Turing machine that decides its membership, or***
- ***reduce it to a set with a solvable membership problem.***

***To show that a set has an unsolvable membership problem:***

- ***prove that no Turing machine decides it's membership, or***
- ***reduce a set with an unsolvable membership problem to it.***

To continue with our discussion about Turing machines and computing membership in certain sets we bring up *properties* of sets at this time. Here is a table containing a few properties and sets of Turing machine indices that correspond to the computable sets possessing that property.

<u>Property</u>	<u>Set defined from the Property</u>
Emptiness	$\{ i \mid W_i = \emptyset \}$
Finiteness	$\{ i \mid W_i \text{ is finite} \}$
Cofiniteness	$\{ i \mid \bar{W}_i \text{ is finite} \}$
Cardinality	$\{ i \mid W_i \text{ has exactly } k \text{ members} \}$
Solvability	$\{ i \mid W_i \text{ is solvable} \}$
Equality	$\{ \langle a, b \rangle \mid W_a = W_b \}$

That was a list of only a few set properties. Note that the set E defined earlier ( $W_i$  contains all integers) was also built from a set property. The set K (machines which halt on their own indices) was not since membership in K depends on the *machine* and not the *set* it accepts.

Speaking about sets, we need to note that there are two *trivial set properties* for the computable sets. One is the property of being computable (all  $W_i$  are in this group since they are defined as exactly those sets accepted by Turing machines) and the other is the property not being computable (no  $W_j$  are in this group of sets).

Here is a major theorem due to Rice that indicates that nothing is solvable when one asks questions about set properties.

**Theorem 2 (Rice).** *Only trivial set properties are solvable for the computable sets.*

**Proof.** The trivial properties (being or not being a computable set) are indeed easily solvable for the class of computable sets because:

$$\begin{aligned} \{ i \mid W_i \text{ is computable} \} &= \{ \text{all positive integers} \} \\ \{ i \mid W_i \text{ is not computable} \} &= \emptyset \end{aligned}$$

and these are easily (trivially) solvable.

We need to show that all of other set properties are unsolvable.

If a property is nontrivial then some set must have the property and some set must not. Let us take an arbitrary property and assume that the set  $W_a$  has this property. Also, the empty set either has the property or does not. Let us assume that it does not. Thus  $W_a$  has the property and  $\emptyset$  does not.

(Note that if  $\emptyset$  had the property the proof would continue along similar lines. This appears later as an exercise in reducing the complement of K to sets.)

Let us give the name P to the set of all indices (from our standard enumeration) of computable sets that possess this property and show that the unsolvable set K is reducible to P. That is, we must find a computable function g such that for all i:

$$i \in K \text{ if and only if } g(i) \in P.$$

Here is how we define  $g$ . For any  $M_i$  we construct the machine:

$$M(x,i) = \begin{cases} M_a(x) & \text{if } M_i(i) \text{ halts} \\ \text{diverge} & \text{otherwise} \end{cases}$$

which operates like  $M_a$  if  $i \in K$  and diverges on all inputs if  $i \notin K$ .

Since all  $M$  does is to run  $M_i(i)$  and then if it halts, turn control over to  $M_a$ ,  $M$  is indeed a Turing machine. Since it is, Church's thesis provides us with an index for  $M$ , and as before, the  $s$ - $m$ - $n$  theorem provides a function  $g$  such that for all  $i$  and  $x$ :

$$M_{g(i)} = M(x,i)$$

So,  $M_{g(i)}$  is a Turing machine in our standard enumeration of Turing machines. Now let us have a look at exactly what  $M_{g(i)}$  accepts. This set is:

$$W_{g(i)} = \begin{cases} W_a & \text{if } i \in K \\ \emptyset & \text{otherwise} \end{cases}$$

because  $M_{g(i)}$  acts exactly like  $M_a$  whenever  $M_i(i)$  halts.

This means that for all  $i$ :

$x \in K$	iff $M_i(i)$ halts	[definition of $K$ ]
	iff $\forall x [M_{g(i)}(x) = M_a(x)]$	[definition of $M_{g(i)}$ ]
	iff $W_{g(i)} = W_a$	[ $M_{g(i)}$ accepts $W_{g(i)}$ ]
	iff $g(i) \in P$	[ $W_a$ has property $P$ ]

Thus we have reduced  $K$  to  $P$ . Applying the corollary of our last theorem tells us that  $P$  is unsolvable and thus whether a computable set has any nontrivial property is likewise an unsolvable problem.

This is a very surprising and disturbing result. It means that almost everything interesting about the computable sets is unsolvable. Therefore we must be very careful about what we attempt to compute.

To conclude the section we present a brief discussion about unsolvability and computer science. Since Turing machines are equivalent to programs, we know

now that there are many questions that we cannot produce programs to answer. This, of course, is because unsolvable problems are exactly those problems that computer programs cannot solve.

Many of the above unsolvable problems can be stated in terms of programs and computers. For example, the halting problem is:

*Does this program contain an infinite loop?*

and we have shown this to be unsolvable. This means that no compiler may contain a routine that unfailingly predicts looping during program execution.

Several other unsolvable problems concerning programs (some of which are set properties and some of which are not) appear on the following list:

- a) Are these two programs equivalent?*
- b) Will this program halt for all inputs?*
- c) Does this program accept a certain set? (Correctness)*
- d) Will this program halt in  $n^2$  steps for an input of length  $n$ ?*
- e) Is there a shorter program that is equivalent to this one?*
- f) Is this program more efficient than that one?*

In closing we shall be so bold as to state the sad fact that almost all of the interesting questions about programs are unsolvable!