11 Using Reductions to Show that Problems are Not Computable

Section 10 provides two tools, Turing reductions and mapping reductions, that we can use to demonstrate that a problem is uncomputable. They are generally much easier to apply than diagonalization. Here are the important facts about reductions from Section 10.

Corollary 10.2. If $A \le_t B$ and A is not computable, then B is not computable.

Corollary 10.4. If $A \le_m B$ and A is not computable then B is not computable.

11.1 Run$(p, x)\uparrow$?

Section 10 defines

$$\begin{aligned} \text{NOTHLT} &= \{(p,x) \mid \text{Run}(p,x)\uparrow\} \\ \text{HLT} &= \{(p,x) \mid \text{Run}(p,x)\downarrow\} \end{aligned}$$

and shows that NOTHLT $\le_t$ HLT. We know from Section 9 that HLT is uncomputable. Relationship NOTHLT $\le_t$ HLT only tells us that NOTHLT is no harder than an uncomputable problem, which tells use nothing about NOTHLT. But it is easy to turn that particular reduction around.

Theorem 11.1. HLT $\le_t$ NOTHLT.

Proof. The following is a Turing reduction from HLT to NOTHLT, establishing that HLT $\le_t$ NOTHLT.

"{halts(p, x):
  If $(p, x) \in$ NOTHLT then
    return 0
  else

```
        return 1
    }"
```

By Corollary 10.2, since HLT is uncomputable, NOTHLT is also uncomputable.

◇———◇

11.2 The acceptance problem

The acceptance problem for programs is as follows.

$$ACC = \{(p, x) \mid \mathrm{Run}(p, x) \cong 1\}.$$

Theorem 11.2. ACC is uncomputable.

Proof. It suffices to show that HLT $\leq_t$ ACC. Here is a Turing reduction from HLT to ACC. It introduces a new wrinkle: it builds a program on the fly.

```
"{halts(p, x):
    r = "{r(z): w = Run(p, z); return 1}"
    if (r, x) ∈ ACC
        return 1
    else
        return 0
}"
```

Clearly

$$\begin{aligned}
(r, x) \in ACC &\leftrightarrow \mathrm{Run}(r, x) \cong 1 && \text{by the definition of ACC}\\
&\leftrightarrow \mathrm{Run}(p, x)\downarrow && \text{by the definition of } r\\
&\leftrightarrow (p, x) \in HLT
\end{aligned}$$

so program halts(p, x) correctly answers the question, is $(p, x) \in$ HLT.

◇———◇

There is really no need for the full power of a Turing reduction here. Function

$$f(p, x) = (\texttt{"}\{r(z) : w = \text{Run}(p, z); \text{return } 1\}\texttt{"}, x)$$

is a mapping reduction from HLT to ACC; f is computable and, as we have just shown,

$$(p, x) \in HLT \leftrightarrow f(p, x) \in ACC.$$

11.3 Does p terminate on input 1?

We have seen the trick of creating a program on the fly. With the next reduction, we introduce another trick: make that program ignore its parameter, so that it does the same thing on all strings. Define

$$T_1 = \{r \mid \text{Run}(r, 1)\downarrow\}.$$

That is, instead of asking whether a give program halts on some given string x, T_1 asks whether the program halts on input 1. That might sound easier than the Halting Problem, but it is not.

Theorem 11.3. T_1 is uncomputable.

Proof. It suffices to show that there is a mapping reduction from HLT to T_1; that is, we show that HLT $\leq_m T_1$. The usual way to show that something exists is to produce one; here is a mapping reduction f from HLT to T_1.

$$f(p, x) = \texttt{"}\{r(q) : w = \text{Run}(p, x);\ \text{return}\ \ 1\}\texttt{"}.$$

Certainly, f is computable. All it does is write a program (a string) and return that program. f does not run the program that it builds. Notice that program $r(q)$ runs program p on input x, but ignores the result. Also notice that $r(q)$ ignores q; r does the same thing regardless of the parameter that is passed to it.

Let's refer to program $\texttt{"}\{r(q) : w = \text{Run}(p, x);\ \text{return } 1\}\texttt{"}$ as $r_{p,x}$, acknowledging the fact that p and x are built into r, and you cannot write $r_{p,x}$ until you know what p and x are. Notice that

$$\begin{array}{rcll}
(p, x) \in \text{HLT} & \to & \text{Run}(p, x)\downarrow & \text{by the definition of HLT} \\
& \to & \text{Run}(r_{p,x}, q)\downarrow \text{ for every } q & \text{by the definition of } r_{p,x} \\
& \to & \text{Run}(r_{p,x}, 1)\downarrow & \\
& \to & r_{p,x} \in T_1 & \text{by the definition of } T_1
\end{array}$$

and

$$
\begin{aligned}
r_{p,x} \in T_1 &\rightarrow \text{Run}(r_{p,x}, 1)\downarrow && \text{by the definition of } T_1 \\
&\rightarrow \text{Run}(p, x)\downarrow && \text{by the definition of } r_{p,x} \\
&\rightarrow (p, x) \in \text{HLT}
\end{aligned}
$$

Putting those together:

$$(p, x) \in \text{HLT} \leftrightarrow r_{p,x} \in T_1.$$

Since $f(p, x) = r_{p,x}$, that is exactly the requirement for f to be a mapping reduction from HLT to T_1.

◇————————————————————◇

11.4 Does p terminate on input 2?

Define

$$T_2 = \{r \mid \text{Run}(r, 2)\downarrow\}.$$

It should be obvious how to modify the proof of Theorem 11.3 to show that HLT $\leq_m T_2$. But we already know that T_1 is uncomputable, so showing that $T_1 \leq_m T_2$ is enough to show that T_2 is uncomputable. Let's do that.

Theorem 11.4. $T_1 \leq_m T_2$.

Proof. All we need to do is to transform a question of whether a program a halts on input 1 into an equivalent question of whether another program b_a halts on input 2. That is easy to do: define

$$b_a = \texttt{"}\{b(q): \text{ return Run}(a, 1)\}\texttt{"}.$$

Clearly,

$$\text{Run}(a, 1)\downarrow \; \leftrightarrow \text{Run}(b_a, 2)\downarrow.$$

That is,

$$a \in T_1 \leftrightarrow b_a \in T_2.$$

So $f(a) = b_a$ is mapping reduction from T_1 to T_2.

◇————————————————————◇

11.5 The everything problem for programs

Define

$$\text{ALL} = \{p \mid \forall x(\text{Run}(p,x){\downarrow})\}.$$

That is, ALL is the following decision problem.

> **Input.** Program p.
>
> **Question.** Does p halt on every input?

Theorem 11.5. ALL is uncomputable.

Proof. It certainly is not enough to argue that an algorithm to solve ALL would need to try every input. That is nonsense. Write a program that clearly halts on every input, such as the following.

```
"{stopper(x)
   return 1
}"
```

Do you need to try it on every input to be sure that it stops on every input? Of course not. Write another program that clearly loops forever on all inputs, such as the following.

```
"{looper(x)
   while(1)
      do nothing
}"
```

You can see from the structure of the program that it loops forever on all inputs. What we need to show is that there is no program R that takes *any* program p as an input and tells you whether p stops on all inputs.

The proof is a mapping reduction from T_1 to ALL. Define

$$\begin{aligned} r_p &= \text{"}\{r(q):\ \text{return Run}(p,1)\}\text{"} \\ f(p) &= r_p \end{aligned}$$

Since r_p ignores its parameter q, it should be clear from the definition of r_p that

$$\text{Run}(p,1){\downarrow} \leftrightarrow \forall q(\text{Run}(r_p,q){\downarrow}).$$

That is,

$$p \in T_1 \leftrightarrow r_p \in \text{ALL}$$

which means that $f(p) = r_p$ is a mapping reduction from T_1 to ALL.

◇————————————————————————————◇

11.6 Complementation and computability

Theorem 11.6. Suppose S is a language over alphabet Σ. If S is a computable then $\overline{S}$ is also computable.

Proof. Suppose that program p computes S. That is, p stops on every input and, for every $x \in \Sigma^*$,

$$\text{Run}(p, x) \cong 1 \leftrightarrow x \in S.$$

The following program computes $\overline{S}$ by flipping answers from 1 to 0 and from 0 to 1.

```
"{complement(x):
   if Run(p, x) == 1
      return 0
   else
      return 1
}"
```

In fact, it is obvious that Theorem 11.6 extends to an equivalence.

Theorem 11.7. S is computable if an only if $\overline{S}$ is computable.

11.7 Rice's Theorem

Excluding the proof of Theorem 11.1, you should notice similarities in the above proofs. Can we prove a general theorem so that those theorems all become corollaries of the general theorem? Such a theorem would say something like, "It is not computable to determine whether a program has a property that is based solely on what that program does when you run it." But that is much too vague. The first step is to find a precise definition of what that means.

11.7.1 Definitions

Definition 11.1. Programs p and q are *equivalent* if $\text{Run}(p, x) \cong \text{Run}(q, x)$ for every x. We write $p \approx q$ to mean that p and q are equivalent programs.

Suppose that L is a set of programs over alphabet Σ. Define $\overline{L} = \Sigma^* - L$.

Definition 11.2. L *nontrivial* is $L \neq \{\}$ and $L \neq \Sigma*$. That is, neither L nor $\overline{L}$ is empty.

The following theorem is obvious.

Theorem 11.8. L is nontrivial if and only if $\overline{L}$ is nontrivial.

Definition 11.3. L *respects equivalence* provided, for every pair of equivalent programs p and q, either p and q are both in L or p and q are both in $\overline{L}$.

The following is immediate from Definition 11.3.

Theorem 11.9. L respects equivalence if and only if $\overline{L}$ respects equivalence.

Definition 11.4. Define

$$\text{LOOP} = \texttt{"}\{\text{LOOP}(x) : \text{loop forever}\}\texttt{"}$$

to be a program that loops forever on all inputs.

11.7.2 Rice's Theorem

Our goal is to prove Rice's Theorem, stating that every nontrivial set of programs that respects equivalence is uncomputable. We will do that using a lemma and a corollary to the lemma. (A lemma is a theorem that is proved as a step in proving a more important theorem.)

Lemma 11.10. If L is a nontrivial set of programs that respects equivalence, where $\text{LOOP} \notin L$, then $\text{HLT} \leq_m L$.

Proof.

1. Suppose that L is a nontrivial set of programs that respects equivalence and where LOOP $\notin L$.

Known variables:	L
Know (1):	L is a set of programs.
Know (2):	L is nontrivial.
Know (3):	L respects equivalence.
Know (4):	$LOOP \notin L$.
Goal:	HLT $\leq_m L$.

2. Since L is nontrivial, there must be some program that is a member of L. Ask someone else to provide one. Let's call it Y.

Known variables:	L, Y
Know (1):	L is a set of programs.
Know (2):	L is nontrivial.
Know (3):	L respects equivalence.
Know (4):	$LOOP \notin L$.
Know (5):	$Y \in L$.
Goal:	HLT $\leq_m L$.

3. For any given p and x, define $r_{p,x}$ as follows.

"{$r_{p,x}(z)$:
  w = Run(p, x)
  return $Y(z)$
}"

Known variables:	L, Y, $r_{p,x}$
Know (1):	L is a set of programs.
Know (2):	L is nontrivial.
Know (3):	L respects equivalence.
Know (4):	$LOOP \notin L$.
Know (5):	$Y \in L$.
Goal:	HLT $\leq_m L$.

4. Notice that, for arbitrary p and x,

$$\begin{aligned} (p,x) \in \text{HLT} &\rightarrow \forall z(\text{Run}(r_{p,x}, z) \cong \text{Run}(Y, z)) \\ &\rightarrow r_{p,x} \approx Y \\ &\rightarrow r_{p,x} \in L \qquad \text{since } L \text{ respects equivalence} \end{aligned}$$

Known variables:	L, Y, $r_{p,x}$
Know (1):	L is a set of programs.
Know (2):	L is nontrivial.
Know (3):	L respects equivalence.
Know (4):	$LOOP \notin L$.
Know (5):	$Y \in L$.
Know (6):	$\forall p \forall x((p,x) \in \text{HLT} \rightarrow r_{p,x} \in L)$
Goal:	HLT $\leq_m L$.

5. Also, for arbitrary p and x,

$$\begin{aligned} (p,x) \notin \text{HLT} &\rightarrow \forall z(\text{Run}(r_{p,x}, z)\uparrow) \\ &\rightarrow r_{p,x} \approx \text{LOOP} \\ &\rightarrow r_{p,x} \notin L \qquad \text{since } L \text{ respects equivalence} \end{aligned}$$

Known variables:	L, Y, $r_{p,x}$
Know (1):	L is a set of programs.
Know (2):	L is nontrivial.
Know (3):	L respects equivalence.
Know (4):	$LOOP \notin L$.
Know (5):	$Y \in L$.
Know (6):	$\forall p \forall x((p, x) \in \text{HLT} \rightarrow r_{p,x} \in L)$
Know (7):	$\forall p \forall x((p, x) \notin \text{HLT} \rightarrow r_{p,x} \notin L)$
Goal:	$\text{HLT} \le_m L$.

6. Define function

$$f(p, x) = r_{p,x}.$$

Clearly, f is computable, since it only needs to write down program $r_{p,x}$. Putting facts (6) and (7) together,

$$(p, x) \in \text{HLT} \leftrightarrow r_{p,x} \in L.$$

So f is a mapping reduction from HLT to L.

◇———————————————————————◇

Corollary 11.11. If L is a nontrivial set of programs that respects equivalence, where LOOP $\notin L$, then L is not computable.

Proof. That follows immediately from Lemma 11.10, corollary 10.4and the fact that HLT is uncomputable.

◇———————————————————————◇

Theorem 11.12. **(Rice's Theorem)** If L is a nontrivial set of programs that respects equivalence, then L is not computable.

Proof. There are two cases: either LOOP $\notin L$ or LOOP $\in L$.

If LOOP $\notin L$, then Theorem 11.12 follows immediately from Corollary 11.11.

So consider the case where LOOP $\in$ Ł. Then LOOP $\notin \overline{L}$. By theorems 11.8 and 11.9, $\overline{L}$ is nontrivial and $\overline{L}$ respects equivalence. So $\overline{L}$ meets the requirements of Corollary 11.11. We conclude that, in this case, $\overline{L}$ is uncomputable. By Theorem 11.7, L is also uncomputable.

11.8 Examples of using Rice's theorem

11.8.1 Example: T_1 is uncomputable

Recall that we defined

$$T_1 = \{r \mid \text{Run}(r, 1)\downarrow\}.$$

Let's reprove that T_1 is uncomputable using Rice's Theorem.

Theorem 11.13. T_1 is uncomputable.

Proof. Since some programs halt on input 1 and some don't, T_1 is nontrivial. Suppose that p and q are two equivalent programs. Then

$$\begin{array}{rcll} p \in T_1 & \leftrightarrow & \text{Run}(p, 1)\downarrow & \text{by the definition of } T_1 \\ & \leftrightarrow & \text{Run}(q, 1)\downarrow & \text{since } p \approx q \\ & \leftrightarrow & q \in T_1 & \text{by the definition of } T_1 \end{array}$$

By Rice's Theorem, T_1 is uncomputable.

11.8.2 Example: is $L(p)$ finite?

Define

$$\text{FINITE} = \{p \mid L(p) \text{ is a finite set}\}.$$

FINITE is the following decision problem.

> **Input.** A program p.
>
> **Question.** Is $L(p)$ finite? That is, is $\{x \mid \text{Run}(p, x) \cong 1\}$ a finite set?

Notice that FINITE is not a finite set! For every computable set S, there are infinitely many programs that solve S. (You can make infinitely many variations on a program without changing the set that it decides.) So there are infinitely many programs p where $L(p) = \{\}$, and all of those are members of FINITE.

Theorem 11.14. FINITE is uncomputable.

Proof. FINITE is nontrivial. Some programs answer 1 on only finitely many inputs, and some answer 1 on infinitely many inputs.

Suppose that p and q are two equivalent programs. Then

$$\begin{aligned} p \in \text{FINITE} &\leftrightarrow L(p) \text{ is a finite set} \\ &\leftrightarrow L(q) \text{ is a finite set} \qquad \text{since } p \approx q \\ &\leftrightarrow q \in FINITE \end{aligned}$$

So FINITE respects equivalence. By Rice's Theorem, FINITE is uncomputable.

◇————————————————————————◇

11.8.3 Example: is $L(p) = \{\}$?

Define

$$\text{EMPTY} = \{p \mid L(p) = \{\}\}.$$

EMPTY is not an empty set! It is the following decision problem.

> **Input.** A program p.
>
> **Question.** Is it the case that $L(p)$ $\{\}$? That is, are there no inputs x on which p stops and answers 1?

Theorem 11.15. EMPTY is uncomputable.

Proof. EMPTY is clearly nontrivial. It also respects equivalence.

$$\begin{aligned} p \in \text{EMPTY} &\leftrightarrow L(p) = \{\} \\ &\leftrightarrow L(q) = \{\} \qquad \text{since } p \approx q \\ &\leftrightarrow q \in EMPTY \end{aligned}$$

By Rice's Theorem, EMPTY is uncomputable.

$\diamond$——————————————————————————$\diamond$

11.9 Are p and q equivalent?

Define

$$\text{EQUIV} = \{(p, q) \mid p \approx q\}.$$

Rice's Theorem has nothing to say about EQUIV because EQUIV is not a set of programs. It is a set of ordered pairs of programs. Nevertheless, we can show that EQUIV is uncomputable.

Theorem 11.16. EQUIV is uncomputable.

Proof. Define

$$\text{NEVERHALT} = \{p \mid \forall x(\text{Run}(p, x)\uparrow).$$

Rice's theorem does tell us that NEVERHALT is uncomputable. Notice that

$$NEVERHALT = \{p \mid p \approx \text{LOOP}\}.$$

Function f defined by

$$f(p) = (p, \text{LOOP})$$

is a mapping reduction from NEVERHALT to EQUIV, since

$$\begin{aligned} p \in NEVERHALT &\leftrightarrow p \approx \text{LOOP} \\ &\leftrightarrow (p, \text{LOOP}) \in \text{EQUIV} \end{aligned}$$

11.10 K

Define

$$K = \{p \mid \text{Run}(p, p) \not\equiv \bot\}.$$

K is a set of programs, but it does not respect equivalence. Let's try to show that K respects equivalence to see where the proof breaks down.

$$\begin{aligned} p \in K &\leftrightarrow \text{Run}(p, p)\downarrow \\ &\leftrightarrow \text{Run}(q, p)\downarrow \quad \text{since } p \approx q \end{aligned}$$

But what q does on input p is irrelevant to determining whether $q \in K$. All that matters is what q does on input q.

Nevertheless, we can show:

Theorem 11.17. K is uncomputable.

Proof. Rice's theorem is not a help here. But it suffices to show that HLT $\leq_m$ K. For arbitrary p and x, define $r_{p,x}$ as follows.

$= $ "{$r_{p,x}(z)$:
  $w = $ Run(p, x)
  return 1
}"

Notice that $r_{p,x}$ ignores its parameter, z. It is evident that

$$\begin{array}{rcl} (p,x) \in HLT & \to & \text{Run}(p,x)\downarrow \\ & \to & \forall z(\text{Run}(r_{p,x}, z)\downarrow) \\ & \to & \text{Run}(r_{p,x}, r_{p,x})\downarrow) \\ & \to & r_{p,x} \in K \end{array}$$

and

$$\begin{array}{rcl} (p,x) \notin HLT & \to & \text{Run}(p,x)\uparrow \\ & \to & \forall z(\text{Run}(r_{p,x}, z)\uparrow) \\ & \to & \text{Run}(r_{p,x}, r_{p,x})\uparrow) \\ & \to & r_{p,x} \notin K \end{array}$$

which tells us that

$$f(p,x) = r_{p,x}$$

is a mapping reduction from HLT to K.

11.11 Concrete examples

Without concrete examples, it can be easy to believe that our theorems about problems being uncomputable are only of abstract, mathematical significance, and have no bearing on the real world. So let's look at some concrete examples to see that the real world is not immune to mathematical theorems.

11.11.1 The 3n+1 problem

The $3n+1$ problem concerns an infinite collection of sequences of integers. Select a positive integer n to start a sequence. Follow it by $n/2$ if n is even and by $3n+1$ if n is odd. Stop the sequence at 1. The sequence starting with 9 is (9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1).

It is not obvious that the $3n+1$ sequence stops for all starting values. It is conceivable that it gets into a cycle. It is also conceivable that, for some starting values, the numbers in the $3n+1$ sequence keep getting larger and larger, without bound. In fact, nobody knows whether every $3n+1$ sequence is finitely long. But we can always make a conjecture.

Conjecture 11.1. The $3n+1$ sequence is finitely long for every start value.

But look at the following program.

```
"{test(n)
   i = n
   while i > 1
      if i is even
         i = i/2
      else
         i = 3i + 1
}"
```

Can you tell whether test(n) is in ALL? If it is, then Conjecture 11.1 is true. If not, then Conjecture 11.1 is false.

11.11.2 Does program p test whether a number is prime?

Now suppose that you are serving as a grader for a computer programming course. One of the assignments for that course asks students to write a program that reads an input $n > 1$ and tells whether n is prime. As grader, you are tasked with determining whether each submission is correct, with the sole criterion for correctness being that the program correctly determines whether n is prime for every integer n. (In the programming language being used, integers can be arbitrarily large, so you can't try the program on a finite range of integers to decide whether it works.)

To make sure that you are ready, you write your own program p to tell if a number is prime. Now, given a student submission q, the problem is to determine whether $q \approx p$. But that is uncomputable! Could that possibly be a problem? Suppose that a particularly devious student submits the following program.

```
"{q(n)
  i = n
  while i > 1
    if i is even
      i = i/2
    else
      i = 3i + 1
  i = 2
  while i < n
    if n mod i == 0
      return 0
    i = i + 1
  return 1
}"
```

You notice that, if Conjecture 11.1 is true, the submitted program q is correct. But if Conjecture 11.1 is false, then there are values n on which q loops forever, meaning that q is incorrect. In order to grade q according to the grading criterion, you must determine whether Conjecture 11.1 is true!

11.11.3 Goldbach's conjecture

The following conjecture is due to Goldbach.

Conjecture 11.2 Every even integer that is greater than 2 is the sum of two prime integers.

For example, $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5$, $10 = 5 + 5$, etc. Nobody knows whether Goldbach's conjecture is true, and it appears to be a very difficult nut to crack. But we can write the following program, which contains an infinite loop that checks, for each even number n, whether there are two prime numbers whose sum is n. If it finds an even number n that is not the sum of two prime numbers, it stops. Otherwise, it loops forever.

```
"{goldbach()
   n = 4
   while 1
      i = 2
      found = 0
      while not found and i < n
         if i is prime and n − i is prime
            found = 1
         i = i + 1
      if not found
         return 0
      n = n + 2
}"
```

To answer Goldbach's conjecture, all you need to do is ask whether program goldbach ever stops. You can ask whether it is in ALL or in T_1 or in a variety of languages because goldbach ignores its input.

prev **next**