Recursively Enumerable Languages, Turing Machines, and Decidability

1 Problem Reduction: Basic Concepts and Analogies

The concept of problem reduction is simple at a high level. You simply take an algorithm that solves one problem and use it as a subroutine to solve another problem. For example, suppose we have two TM's: C, which turns ❑w❑ into ❑w❑w❑ and S_L, which turns …❑w❑ into …w❑❑ (i.e., it shifts w one square to the left). Then we can build a TM M' that computes f(w) = ww by simply letting M' =CS_L. We have reduced the problem of computing f to the problem of copying a string and then shifting it.

Let's consider another example. Suppose we had a function sqr(m: integer): integer, which accepts an integer input m and returns m^2. Then we could write the following function to compute $g(m) = m^2 + 2m + 1$:

```
function g(m: integer): integer;
begin
     return sqr(m + 1);
end;
```

We have reduced the problem of computing $m^2 + 2m + 1$ to the problem of computing m + 1 and squaring it.

We are going to be using reduction specifically for decision problems. In other words, we're interested in a particular class of boolean functions whose job is to look at strings and tell us, for each string, whether or not it is in the language we are concerned with. For example, suppose we had a TM M that decides the language a^*b^+. Then we could make a new TM M' to decide a*b*: M' would find the first blank to the right of its input and write a single b there. It would then move its read head back to the beginning of the string and invoke M on the tape. Why does this work? Because $x \in a^*b^*$ iff $xb \in a^*b^+$. Looking at this in the more familiar procedural programming style, we are saying that if we have a function: f1(x: string): boolean, which tells us whether $x \in a^*b^+$, then we could write the following function that will correctly tell us whether $x \in a^*b^*$.

```
function f2(x: string): boolean;
begin
    return f1(x || 'b');
end;
```

If, for some reason, you believed that a*b* were an undecidable language, then this reduction would force you to believe that a^*b^+ is also undecidable. Why? Because we have shown that we can decide a*b* provided only that f1 does in fact decide a^*b^+. If we know that we can't decide a*b*, there must be something standing in the way. Unless we're prepared to believe that subroutine invocation is not computable or that concatenation of a single character to the end of a string is not computable, we must assign blame to f1 and conclude that we didn't actually have a correct f1 program to begin with.

These examples should all make sense. The underlying concept is simple. Things will become complicated in a bit because we will begin considering TM descriptions as the input stings about which we want to ask questions, rather than simple strings of a's and b's.

Sometimes, we'll use a slightly different but equivalent way of asking our question. Rather than asking whether a language is decidable, we may ask whether a problem is solvable. When we say that a problem is unsolvable, what we mean is that the corresponding language is undecidable. For example, consider the problem of determining, given a TM M and string w, whether or not M accepts w. We can ask whether this problem is solvable. But notice that this same problem can be phrased a language recognition problem because it is equivalent to being able to decide the language:

$H = \{\text{"M" "w"} : w \in L(M)\}$.

Read this as: H is the language that consists of all strings that can be formed from two parts: the encoding of a Turing

Machine M, followed by the encoding of a string w (which we can think of as the input to M), with the additional constraint that TM M halts on input w (which is equivalent to saying that w is in the language accepted by M).

"Solving a problem" is the higher level notion, which is commonly used in the programming/algorithm context. In our theoretical framework, we use the term "deciding a language" because we are talking about Turing Machines, which operate on strings, and we have a carefully constructed theory that lets us talk about Turing Machines as language recognizers.

In the following section, we'll go through several examples in which we use the technique of problem reduction to show that some new problem is unsolvable (or, alternatively, that the corresponding language is undecidable). All of these proofs depend ultimately on our proof, using diagonalization, of the undecidability of the halting problem (H above).

2 Some Reductions Presented in Gory Detail

Example 1: Given a TM M, does M halt on input ε? (i.e., given M, is $\varepsilon \in L(M)$?) This problem is undecidable because we can reduce the Halting Problem H to it. What this means is that an algorithm to answer this question could be used as a subroutine in an algorithm (which is otherwise clearly effective) to solve the Halting problem. But we know the Halting problem is unsolvable; therefore this question is unsolvable. So how do we prove this?

First we'll prove this using the TM/language framework. In other words, we're going to show that the following language LE is not decidable:

LE = {"M": $\varepsilon \in L(M)$}

We will show that if LE is decidable, so is H = {"M" "w" : $w \in L(M)$}.

Suppose LE is decidable; then some TM M_{LE} decides it. We can now show how to construct a new Turing Machine M_H, which will invoke M_{LE} as a subroutine, and which will decide H. In the process of doing so, we'll use only clearly computable functions (other than M_{LE}, of course). So when we finish and realize that we have a contradiction (since we know that M_H can't exist), we know that the blame must rest on M_{LE} and thus we know that M_{LE} cannot exist.

M_H is the Turing Machine that operates as follows on the inputs "M", "w":

1. Construct a new TM M*, which behaves as follows:
   1.1. Copy "w" onto its tape.
   1.2. Execute M on the resulting tape.
2. Invoke M_{LE}(M*).

If M_{LE} returns True, then we know that M (the original input to M_H) halts on w. If M_{LE} returns False, then we know that it doesn't. Thus we have built a supposedly unbuildable M_H. How did we do it? We claimed when we asserted the existence of M_{LE} that we could answer what appears to be a more limited question, does M halt on the empty tape? But we can find out whether M halts on any other specific input (w) by constructing a machine (M*) that starts by writing w on top of whatever was originally on its tape (thus it ignores its actual input) and then proceeds to do whatever M would have done. Thus M* behaves the same on all inputs. Thus if we knew what it does on any one input, we'd know what it does for all inputs. So we ask M_{LE} what it does on the empty string. And that tells us what it does all the time, which must be, by the way we constructed it, whatever the original machine M does on w.

The only slightly tricky thing here is the procedure for constructing M*. Are we sure that it is in fact computable? Maybe we've reached the contradiction of claiming to have a machine to solve H not by erroneously claiming to have M_{LE} but rather by erroneously claiming to have a procedure for constructing M*. But that's not the case. Why not? It's easy to see how to write a procedure that takes a string w and builds M*. For example, if "w" is "ab", then M* must be:

ERaRbL$_{\sqcup}$M, where E is a TM that erases its tape and then moves the read head back to the first square.

In other words, we erase the tape, move back to the left, then move right one square (leaving one blank ahead of the new tape contents), write a, move right, write b, move left until we get back to the blank that's just to the left of the input, and then execute M.

The Turing Machine to construct M* is a bit too complicated to write here, but we can see how it works by describing it in a more standard procedural way: It first writes ER. Then, for each character in w, it writes that character and R. Finally it writes $L_{\sqcup}M$.

To make this whole process even clearer, let's look at this problem not from the point of view of the decidability of the language H but rather by asking whether we can solve the Halting problem. To do this, let's describe in standard code how we could solve the Halting problem if we had a subroutine M_{LE}(M: TM) that could tell us whether a particular Turing Machine halts on the empty string. We'll assume a datatype TM. If you want to, you can think of objects of this type as essentially strings that correspond to valid Turing Machines. It's like thinking of a type Cprogram, which is all the strings that are valid C programs.

We can solve the Halting program with following function Halts:

```
Function Halts(M: TM, w: string): Boolean;
    M* := Construct(M, w);
    Return MLE(M*);
    end;

Function Construct(M: TM, w: string): TM;
    /* Construct builds a machine that first erases its tape.  Then it copies w onto its tape and moves its
    /* read head back to the left ready to begin reading w.  Finally, it executes M.
    Construct := Erase;          /* Erase is a string that corresponds to the TM that erases its input tape.
    For each character c in w do
        Construct := Construct || "R" || c;
        end;
    Construct := Construct || L␣M;
    Return(Construct);

Function MLE(M: TM): Boolean;
    The function we claim tells us whether M halts on the empty string.
```

The most common mistake that people make when they're trying to use reduction to show that a problem isn't solvable (or that a language isn't decidable) is to do the reduction backwards. In this case, that would mean we would put forward the following argument: "Suppose we had a program Halts to solve the Halting problem (the general problem of determining whether a TM M halts on some arbitrary input w). Then we could use it as a subroutine to solve the specific problem of determining whether a TM M halts on the empty string. We'd simply invoke Halts and pass it M and ε. If Halts returns True, then we say yes. If Halts returns False, we say no. But since we know that Halts can't exist, no solution to our problem can exist either." The flaw in this argument is the last sentence. Clearly, since Halts can't exist, this particular approach to solving our problem won't work. But this says nothing about whether there might be some other way to solve our problem.

To see this flaw even more clearly, let's consider applying it in a clearly ridiculous way: "Suppose we had a program Halts to solve the Halting problem. Then we could use it as a subroutine to solve the problem of adding two numbers a and b. We'd simply invoke Halts and pass it the trivial TM that simply halts immediately and the input ε. If Halts returns True, then we return a+b. If Halts returns False, we also return a+b. But since we know that Halts can't exist, no solution to our problem can exist either." Just as before, we have certainly written a procedure for adding two numbers that won't work, since Halts can't exist. But there are clearly other ways to solve our problem. We don't need Halts. That's totally obvious here. It's less so in the case of attempting to build M_{LE}. But the logic is the same in both cases: flawed.

Example 2: Given a TM M, is $L(M) \neq \varnothing$? (In other words, does M halt on anything at all?). Let's do this one first using the solvability of the problem perspective. Then we'll do it from the decidability of the language point of view.

This time, we claim that there exists:

```
Function MLA(M: TM): Boolean;
    Returns T if M halts on any inputs at all and False otherwise.
```

We show that if this claim is true and MLA does in fact exist, then we can build a function MLE that solves the problem of determining whether a TM accepts the empty string. We already know, from our proof in Example 1, that this problem isn't solvable. So if we can do it using MLA (and some set of clearly computable functions), we know that MLA cannot in fact exist.

The reduction we need for this example is simple. We claim we have a machine MLA that tells us whether some machine M accepts anything at all. If we care about some particular input to M (for example, we care about ε), then we will build a new machine M* that erases whatever was originally on its tape. Then it copies onto its tape the input we care about (i.e., ε) and runs M. Clearly this new machine M* is oblivious to its actual input. It either always accepts (if M accepts ε) or always rejects (if M rejects ε). It accepts everything or nothing. So what happens if we pass M* to MLA? If M* always accepts, then its language is not the empty set and MLA will return True. This will happen precisely in case M halts on ε. If M* always rejects, then its language is the empty set and MLA will return False. This will happen precisely in case M doesn't halt on ε. Thus, if MLA really does exist, we have a way to find out whether any machine M halts on ε:

```
Function MLE(M: TM): Boolean;
    M* := Construct(M);
    Return MLA(M*);
    end;

Function Construct(M: TM): TM;  /* This time, we build an M* that simply erases its input and then runs M
                                /*(thus running M on ε).
    Construct := Erase;         /* Erase is a string that corresponds to the TM that erases its input tape.
    Construct := Construct || M;
    Return(Construct);
```

But we know that MLE can't exist. Since everything in its code, with the exception of MLA, is trivially computable, the only way it can't exist is if MLA doesn't actually exist. Thus we've shown that the problem determining whether or not a TM M halts on any inputs at all isn't solvable.

Notice that this argument only works because everything else that is done, both in MLE and in Construct is clearly computable. We could write it all out in the Turing Machine notation, but we don't need to, since it's possible to prove that anything that can be done in any standard programming language can also be done with a Turing Machine. So the fact that we can write code for it is good enough.

Whenever we want to try to use this approach to decide whether or not some new problem is solvable, we can choose to reduce to the new problem any problem that we already know to be unsolvable. Initially, the only problem we knew wasn't solvable was the Halting problem, which we showed to be unsolvable using diagonalization. But once we have used reduction to show that other problems aren't solvable either, we can use any of them for our next problem. The choice is up to you. Whatever is easiest is the thing to use.

When we choose to use the problem solvability perspective, there is always a risk that we may make a mistake because we haven't been completely rigorous in our definition of the mechanisms that we can use to solve problems. One big reason for even defining the Turing Machine formalism is that it is both simple and rigorously defined. Thus, although the problem solvability perspective may seem more natural to us, the language decidability perspective gives us a better way to construct rigorous proofs.

So let's show that the language

$LA = \{\text{"M"}: L(M) \neq \varnothing\}$ is undecidable.

We will show that if LA were decidable, then $LE = \{\text{"M"}: \varepsilon \in L(M)\}$ would also be decidable. But of course, we know that it isn't.

Suppose LA is decidable; then some TM M_{LA} decides it. We can now show how to construct a new Turing Machine M_{LE},

which will invoke M_{LA} as a subroutine, and which will decide LE:

M_{LE}(M): /* A decision procedure for LE = {"M": ε ∈ L(M)}
1. Construct a new TM M*, which behaves as follows:
   1.1. Erase its tape.
   1.2. Execute M on the resulting empty tape.
2. Invoke M_{LA}(M*).

It's clear that M_{LE} effectively decides LE (if M_{LA} really exists). Why? M_{LE} returns True iff M_{LA} returns True. That happens, by its definition, if it is passed a TM that accepts at least one string. We pass it M*. M* accepts at least one string (in fact, it accepts all strings) precisely in case M accepts the empty string. If M does not accept the empty string, then M* accepts nothing and M_{LE} returns False.

Example 3: Given a TM M, is L(M) = Σ*? (In other words, does M accept everything?). We can show that this problem is unsolvable by using almost exactly the same technique we just used. In example 2, we wanted to know whether a TM accepted anything at all. Now we want to know whether it accepts everything. We will answer this question by showing that the language

LΣ = {"M": L(M) = Σ*} is undecidable.

Recall the machine M* that we constructed for Example 2. It erases its tape and then runs M on the empty tape. Clearly M* either accepts nothing or it accepts everything, since its behavior is independent of its input. M* is exactly what we need for this proof too. Again we'll choose to reduce the language LE = {"M": ε ∈ L(M)} to our new problem L:

If LΣ is decidable, then there's a TM $M_{L\Sigma}$ that decides it. In other words, there's a TM that tells us whether or not some other machine M accepts everything. If $M_{L\Sigma}$ exists, then we can define the following TM to decide LE:

M_{LE}(M): /* A decision procedure for LE = {"M": ε ∈ L(M)}
1. Construct a new TM M*, which behaves as follows:
   1.1. Erase its tape.
   1.2. Execute M on the resulting empty tape.
2. Invoke $M_{L\Sigma}$(M*).

Step 2 will return True if M* halts on all strings in Σ* and False otherwise. So it will return True if and only M halts on ε. This would seem to be a correct decision procedure for LE. But we know that such a procedure cannot exist and the only possible flaw in the procedure we've given is $M_{L\Sigma}$. So $M_{L\Sigma}$ doesn't exist either.

Example 4: Given a TM M, is L(M) infinite? Again, we can use M*. Remember that M* either halts on nothing or it halts on all elements of Σ*. Assuming that Σ ≠ ∅, that means that M* either halts on nothing or it halts on an infinite number of strings. It halts on everything if its input machine M halts on ε. Otherwise it halts on nothing. So we can show that the language

LI = {"M": L(M) is infinite}

is undecidable by reducing the language LE = {"M": ε ∈ L(M)} to it.

If LI is decidable, then there is a Turing Machine M_{LI} that decides it. Given M_{LI}, we decide LE as follows:

M_{LE}(M): /* A decision procedure for LE = {"M": ε ∈ L(M)}
1. Construct a new TM M*, which behaves as follows:
   1.1. Erase its tape.
   1.2. Execute M on the resulting empty tape.
2. Invoke M_{LI}(M*).

Step 2 will return True if M* halts on an infinite number of strings and False otherwise. So it will return True if and only M halts on ε.

This idea that a single construction may be the basis for several reduction proofs is important. It derives from the fact that several quite different looking problems may in fact be distinguishing between the same two cases.

Example 5: Given two TMs, M_1 and M_2, is $L(M_1) = L(M_2)$? In other words, is the language
LEQ = {"M_1" "M_2": $L(M_1) = L(M_2)$} decidable?

Now, for the first time, we want to answer a question about the relationship of two Turing Machines to each other. How can we solve this problem by reducing to it any of the problems we already know to be undecidable? They all involve only a single machine. The trick is to use a constant, a machine whose behavior we are certain of. So we define M#, which halts on all inputs. M# is trivial. It ignores its input and goes immediately to a halt state.

If LEQ is decidable, then there is a TM M_{LEQ} that decides it. Using M_{LEQ} and M#, we can decide the language
$L\Sigma$ = {"M": $L(M) = \Sigma^*$} (which we showed in example 3 isn't decidable) as follows:

$M_{L\Sigma}$(M): /* A decision procedure for $L\Sigma$
1. Invoke M_{LEQ}(M, M#).

Clearly M accepts everything if it is equivalent to M#, which is exactly what M_{LEQ} tells us.

This reduction is an example of an easy one. To solve the unsolvable problem, we simply pass the input directly into the subroutine that we are assuming exists, along with some simple constant. We don't need to do any clever constructions. The reason this was so simple is that our current problem LEQ, is really just a generalization of a more specific problem we've already shown to be unsolvable. Clearly if we can't solve the special case (determining whether a machine is equivalent to M#), we can't solve the more general problem (determining whether two arbitrary machines are equivalent).

Example 6: Given a TM M, is L(M) regular? Alternatively, is
LR = {"M": L(M) is regular} decidable?
.
To answer this one, we'll again need to use an interesting construction. To do this, we'll make use of the language
H = {"M" "w" : $w \in L(M)$}
Recall that H is just the set of strings that correspond to a (TM, input) pair, where the TM halts on the input. H is not decidable (that's what we proved by diagonalization). But it is semidecidable. We can easily built a TM H_{semi} that halts whenever the TM M halts on input w and that fails to halt whenever M doesn't halt on w. All H_{semi} has to do is to simulate the execution of M on w. Note also that H isn't regular (which we can show using the pumping theorem).

Suppose that, from M and H_{semi}, we construct a machine M$ that behaves as follows: given an input string w, it first runs H_{semi} on w. Clearly, if H_{semi} fails to halt on w, M$ will also fail to halt. But if H_{semi} halts, then we move to the next step, which is to run M on ε. If we make it here, then M$ will halt precisely in case M would halt on ε. So our new machine M$ will either:
1. Accept H, which it will do if $\varepsilon \in L(M)$, or
2. Accept ∅, which it will do if $\varepsilon \notin L(M)$.

Thus we see that M$ will accept either
1. A non regular language, H, which it will do if $\varepsilon \in L(M)$, or
2. A regular language ∅, which it will do if $\varepsilon \notin L(M)$.

So, if we could tell whether M$ accepts a regular language or not, we'd know whether or not M accepts ε.

We're now ready to show that LR isn't decidable. If it were, then there would be some TM M_{LR} that decided it. But M_{LR} cannot exist, because, if it did, we could reduce LE = {"M": $\varepsilon \in L(M)$} to it as follows:

M_{LE}(M): /* A decision procedure for LE = {"M": $\varepsilon \in L(M)$}
1. Construct a new TM M$(w), which behaves as follows:
   1.1. Execute H_{semi} on w.

1.2. Execute M on ε.
2. Invoke M_{LR}(M$).
3. If the result of step 2 is True, return False; if the result of step 2 is False, return True.

$M_{LE,}$ as just defined, effectively decides LE. Why? If $\varepsilon \in$ L(M), then L(M$) is H, which isn't regular, so M_{LR} will say False and we will, correctly, say True. If $\varepsilon \notin$ L, then L(M$) is ∅, which is regular, so M_{LR} will say True and we will, correctly, say False.

By the way, we can use exactly this same argument to show that LC = {"M": L(M) is context free} and LD = {"M": L(M) is recursive} are undecidable. All we have to do is to show that H is not context free (by pumping) and that it is not recursive (which we did with diagonalization).

Example 7: Given a TM M and state q, is there any configuration (p, u<u>a</u>v), with p ≠ q, that yields a configuration whose state is q? In other words, is there any state p that could possibly lead M to q? Unlike many (most) of the questions we ask about Turing Machines, this one is not about future behavior. (e.g., "Will the Turing Machine do such and such when started from here?") So we're probably not even tempted to try simulation (which rarely works anyway).

But there is a way to solve this problem. In essence, we don't need to consider the infinite number of possible configurations of M. All we need to do is to examine the (finite) transition table of M to see whether there is any transition from some state other than q (call it p) to q. If there is such a transition (i..e., if ∃ p, σ, τ such that δ(p, σ) = (q, τ)), then the answer is yes. Otherwise, the answer is no.

3 Rice's Theorem

Rice's Theorem makes a very general statement about an entire class of languages that are not recursive. Thus, for some languages, it is an alternative to problem reduction as a technique for proving that a language is not recursive.

There are several different forms in which Rice's Theorem can be stated. We'll present two of them here:

(Form 1) Any nontrivial property of the recursively enumerable languages is not decidable.

(Form 2) Suppose that C is a proper, nonempty subset of the class of all recursively enumerable languages. Then the following language is undecidable: LC = {<M>: L(M) is an element of C}.

These two statements look quite different but are in fact nearly equivalent. (Although Form 1 is stronger since it makes a claim about the language, no matter how you choose to define the language. So it applies given a grammar, for example. Form 2 only applies directly if the way you define the language is by a Turing Machine that semidecides it. But even this difference doesn't really matter since we have algorithms for constructing a Turing Machine from a grammar and vice versa. So it would just take one more step if we started with a grammar and wanted to use Form 2). But, if we want to prove that a language of Turing machine descriptions is not recursive using Rice's Theorem, you must do the same things, whichever description of it you prefer.

We'll consider Form 1 first. To use it, we first, we have to guarantee that the property we are concerned with is a property (predicate) whose domain is the set of recursively enumerable languages. Here are some properties P whose domains are the RE languages:
1) P is true of any RE language that contains an even number of strings and false of any RE language that contains an odd number of strings.
2) P is true of any RE language that contains all stings over its alphabet and false for all RE languages that are missing any strings over their alphabet.
3) P is true of any RE language that is empty (i.e., contains no strings) and false of any RE language that contains any strings.
4) P is true of any RE language

5) P is true of any RE language that can be semidecided by a TM with an even number of states and false for any RE language that cannot be semidecided by such a TM.
6) P is true of any RE language that contains at least one string of infinite length and false of any RE language that contains no infinite strings.

Here are some properties whose domains are not the RE languages:
1′) P is true of Turing machines whose first move is to write "a" and false of other Turing machines.
2′) P is true of two tape Turing machines and false of all other Turing machines.
3′) P is true of the negative numbers and false of zero and the positive numbers.
4′) P is true of even length strings and false of odd length strings.

We can attempt to use Rice's Theorem to tell us that properties in the first list are undecidable. It won't help us at all for properties in the second list.

But now we need to do one more thing: We must show that P is a *nontrivial* property. Any property P is nontrivial if it is not equivalent to True or False. In other words, it must be true of at least one language and false of at least one language.

Let's look at properties 1-6 above:
1) P is true of {"a", "b"} and false of {"a"}, so it is nontrivial.
2) Let's just consider the case in which Σ is {a, b}. P is true of Σ* and false of {"a"}.
3) P is true of $\varnothing$ and P is false of every other RE language.
4) P is true of any RE language and false of nothing, so P is trivial.
5) P is true of any RE language and false of nothing, so P is trivial. Why? Because, for any RE language L there exists a semideciding TM M. If M has an even number of states, then P is clearly true. If M has an odd number of states, then create a new machine M′ identical to M except it has one more state. This state has no effect on M′s behavior because there are no transitions in to it. But it guarantees that M′ has an even number of states. Since M′ accepts L (because M does), P is true of L. So P is true of all RE languages.
6) P is false for all RE languages. Why? Because the definition of a language is a set of strings, each of finite length. So no RE language contains a string of infinite length.

So we can use Rice's theorem to prove that the set of RE languages possessing any one of properties 1, 2, or 3 is not recursive. But it does not tell us anything about the set of RE languages possessing property 3, 4, or 5.

In summary, to apply this version of Rice's theorem, it is necessary to do three things:
0) Specify a property P.
1) Show that the domain of P is the set of recursively enumerable languages.
2) Show that P is nontrivial by showing:
   a) That P is true of at least one language, and
   b) That P is false of at least on language.

Now let's try to use Form 2. We must find a C that is a proper, nonempty subset of the class of all recursively enumerable language.

First we notice that this version is stated in terms of C, a subset of the RE languages, rather than P, a property (predicate) that is true of the RE languages. But this is an insignificant difference. Given a universe U, then one way to define a subset S of U is by a characteristic function that, for any candidate element x, returns true if $x \in S$ and false otherwise. So the P that corresponds to S is simply whatever property the characteristic function tests for. Alternatively, for any subset S there must exist a characteristic function for S (although that function need not be computable – that's a different issue.) So given a set S, we can define the property P as "is a member of S." or "possesses whatever property it takes to be determined to be n S." So Form 2 is stated in terms of the set of languages that satisfy some property P instead of being stated in terms of P directly, but as there is only one such set for any property P and there is only one such property P (viewed simply as a truth table, ignoring how you say it in English) for any set, it doesn't matter which specification we use.

Next we notice that this version requires that C be a proper, nonempty subset of the class of RE languages. But this is exactly the same as requiring that P be nontrivial. Why? For P to be nontrivial, then there must exist at least one language of which it is true and one of which it is false. Since there must exist one language of which it is true, the set of languages that satisfy it isn't empty. Since there must exist one language of which it is false, the set of languages that satisfy it is not exactly the set of RE languages and so we have a proper subset.

So, to use Form 2 of Rice's Theorem requires that we:
0) Specify some set C (by specifying a membership predicate P or some other way).
1) Show that C is a subset of the set of RE languages (which is equivalent to saying that the domain of its membership predicate is the set of RE languages)
2) Show that C is a proper nonempty subset of the recursive languages by showing that
   a) $C \neq \varnothing$ (i.e., its characteristic function P is not trivially false), and
   b) C ≠ RE (i.e., its characteristic function P is not trivially true).