Notes on Effective Computability

Robert M. Keller
4 December 2006

We assume that there is a basic understanding of the operation of Turing machines. We are using the definition of Turing machine in Kozen's book, which is not exactly the standard, but all models are ultimately interchangeable, so it will do.

This is Really About Algorithms

Although the topic is presented in the setting of Turing machines, it is really about the existence and non-existence of algorithms. As we know, Turing's thesis was that any algorithmic process can be represented by one of these machines. Particularly in the aspects that use transformations or modifications of Turing machines, those modifications could be carried out in most any general programming language. Moreover, the machines on which they operate could also equally well have been programs in the same or a different language. A programming language essentially performs the same functions as a universal Turing machine.

When terms such as recursive, decidable, solvable, effective, and computable are used, they refer to the existence of algorithms for achieving certain tasks. They are not about a single task and whether it has a yes-no answer, but rather about families of tasks that are handled in a uniform manner by an algorithm. For example, when someone states "the halting problem is unsolvable", the real problem is not answering a question about a specific machine. The problem is the construction of an algorithm that gives an answer for any Turing machine, the encoding of which is supplied as input.

Basic Notation Used Here

Given a string x, a Turing machine M, when started in standard position with x on its tape can be characterized by three mutually-exclusive predicates:

- accept(M, X): M eventually reaches the accept state
- reject(M, X): M eventually reaches the reject state
- loop(M, X): Neither of the above

Note: "Loop" is used colloquially, and does not necessarily mean that the machine repeatedly revisits the same configuration. That is only one form of looping. Looping behavior also includes not ever revisiting the same

configuration again, as long as neither accept nor reject states are reached (per Kozen's conventions). The term "diverge" is sometimes used, and is probably a better alternative to "loop".

We define an additional predicate in terms of the three basic ones:

- halt(M, X) ↔ (accept(M, X) ∨ reject(M, X))

We also note the validity of:

- ¬accept(M, X) ↔ (reject(M, X) ∨ loop(M, X))

To simplify expression of certain ideas, we define a 3-valued function ***outcome*** with symbolic range {*accept*, *reject*, *loop*}:

outcome(M, x) = *accept* if accept(M, x), *reject* if reject(M, x), else *loop*.

Language Accepted by a Turing Machine

The language accepted by a TM is defined as:

$$L(M) = \{x \mid accept(M, x)\}$$

Note that this definition does not say what happens if ¬accept(M, X); unlike a DFA, it could be either reject(M, X) or loop(M, X). So the **complement** of L(M), Σ^* - L(M), where Σ is the input alphabet, is denoted by ~L(M):

$$\sim L(M) = \{x \mid reject(M, x) \vee loop(M, x)\}$$

Let's say that a TM is **total** if it cannot loop, i.e. it always halts. In this case, let's say that the TM **totally accepts** its language.

Being total is not something we'd expect to verify by casual inspection of the TM. It would normally require a proof.

Universal Encoding

Following Kozen, we use M#x to denote the uniform encoding of an arbitrary TM M and its tape x each into {0, 1}*. If we wished, we could even require # ∈ {0, 1}* by reserving its encoding as something that cannot appear as a suffix of an encoded TM. The important thing is that we be able to separate the encoding of M from the encoding of x.

A universal TM U simulates an arbitrary TM M with input x, in the sense that

$$\text{outcome}(U, M\#x) = \text{outcome}(M, x)$$

Existence of Non Turing-Acceptable Languages

Since every TM can be encoded as a string in {0, 1}*, we know that the number of TM's, hence TM-acceptable languages, is countable. Every language can be identified by an infinite vector of 0's and 1's, where the i^{th} element of the vector is 1 if the i^{th} element of {0, 1}* is in the language, just as every real number in the interval [0, 1) can be identified with such a vector. Hence, by the same **diagonalization** argument that the set of real numbers is not countable, the set of all languages is *not* countable. In other words, there are a lot more languages than there are Turing-acceptable ones. The latter are the exception. Shortly, we show some specific languages that are not Turing-acceptable.

Manipulating TM Encodings

In much of the discussion to follow, there will be the need to manipulate the encoding of a machine to create the encoding of a new machine M'. This is usually just a matter of adding some additional 5-tuples that represent new transitions, although it sometimes involves changing the character of states in M, e.g. from accept to something else. Most of these manipulations are described informally, but could be made precisely algorithmic if need be.

Here is a trivial example for illustration:

Observation 1: For an TM M, there is a machine M' such that L(M') = L(M), but M' never rejects (i.e. it either accepts or loops).

In order to construct M', we just change the reject state into an ordinary state. By the convention in Kozen, once that state is entered, it is never left, so rejecting is replaced with looping. We need to add a new reject state for sake of the definition, that every TM has one, but it is not connected to any other state.

Recursive and Recursive-Enumerability

The above terms have a technical meaning in computability. Although they were originally derived from recursive functional programs, that meaning is forgotten, and replaced with the following:

- A language is **recursively-enumerable** (r.e.) iff it is L(M) for some TM M.

- A language is **recursive** iff it is L(M) for some *total* TM M.

If a given language is accepted by one TM, then it is accepted by an infinite number of them (because for any given machine, at a minimum we can add on transitions that "do nothing" in an infinite number of ways). Some of the machines that accept a given language may be total and others not.

Obviously recursive implies recursively-enumerable. It is not immediately obvious, but the converse is not true. There are languages such that a TM can *verify* MemberSelf without being able to *test* MemberSelf.

Observation 2: If a language L is recursive, then its complement ~L is recursive.

Consider a TM M that always halts and accepts L. By interchanging the accepting and rejecting states of M, we have a TM that accepts ~L.

Theorem 1: If a language and its complement are both recursively-enumerable, then the language is recursive.

Suppose that L and ~L are accepted by M and M' respectively. We can combine these machines into a new machine M'' that accepts L and which always halts.

Given an input x, M'' simulates M on x and M' on x in an *interleaved* fashion (e.g. by creating separate tracks in which to simulate each). By assumption, exactly one of M or M' will eventually reach an accepting state. If M reaches it, then x is accepted. If M' reaches it, then x is rejected. So the simulating machine M'' will always halt and accepts L.

Turing Machines Self-Applied

Since machines can be encoded as strings, nothing prevents us from operating on those strings as data. We might try to compute useful predicates, such as:

$$\text{loops}(M\#x) \leftrightarrow \text{loops}(M, x)$$

Such a predicate would be useful in telling us whether the machine is going to loop, so we wouldn't have to wait if we knew that. As useful as they might seem, some predicates of this nature are not computable. Note that a predicate such as loops is entirely equivalent to a language: the language is the set of strings that satisfy the predicate, and for every predicate on strings, there is a language of strings that satisfy it.

Here is a language related to the above function:

$$\text{NonMemberSelf} = \{M \mid \neg\text{accept}(M, M)\}$$

By M as a *string*, we of course mean an *encoding* of machine M.

Theorem 2: NonMemberSelf is not recursively-enumerable.

Proof: Note that for every Turing machine M:

$$M \in \text{NonMemberSelf} \leftrightarrow \neg \text{accept}(M, M) \qquad (1)$$

Suppose that NonMemberSelf is recursively enumerable, i.e. that is there is a TM K such that

$$L(K) = \text{NonMemberSelf} \qquad (2)$$

From (1), we know specifically:

$$K \in \text{NonMemberSelf} \leftrightarrow \neg \text{accept}(K, K) \qquad (3)$$

i.e., from (2):

$$K \in L(K) \leftrightarrow \neg \text{accept}(K, K) \qquad (3')$$

However, by *definition* of the predicate *accept*:

$$K \in L(K) \leftrightarrow \text{accept}(K, K) \qquad (4)$$

So there is a blatant contradiction between (3') and (4), meaning that our supposition was wrong; NonMemberSelf is not recursively-enumerable.

Observation 3: The language

$$\text{MemberSelf} = \{M \mid \text{accept}(M, M)\}$$

Is recursively-enumerable. A machine that accepts MemberSelf can be constructed by appending transitions to a universal Turing Machine. From the input M, the machine creates M#M then behaves exactly as the universal machine.

Observation 4: The language MemberSelf is recursively-enumerable, but not recursive. This follows as an application of Theorem 1.

Here is a language similar to NonMemberSelf:

$$\text{LoopsSelf} = \{M \mid \text{loops}(M, M)\}$$

Theorem 3: LoopsSelf is not recursively-enumerable.

Proof: Note that for every Turing machine M:

$$M \in \text{LoopsSelf} \leftrightarrow \text{loops}(M, M) \qquad (1)$$

Suppose that LoopsSelf is recursively enumerable, that is there is a TM K such that

$$L(K) = \text{LoopsSelf} \qquad (2)$$

From Observation 1, we can modify K to get K' which never rejects. It either accepts or loops, but still L(K') = L(K), thus:

$$L(K') = \text{LoopsSelf} \qquad (2')$$

A special case of (1) is:

$$K' \in \text{LoopsSelf} \leftrightarrow \text{loops}(K', K') \qquad (3)$$

and from (2'), with the fact that K' only accepts or loops:

$$K' \in \text{LoopsSelf} \leftrightarrow \neg\text{loops}(K', K') \qquad (4)$$

Evidently, (3) and (4) are in direct contradiction. Hence our assumption that LoopsSelf is recursively-enumerable was wrong.

The Halting Problem

In the vernacular, creating a machine that accepts LoopsSelf is called "The Halting Problem", and Theorem 3 is often phrased "The Halting Problem" is unsolvable. Perhaps it might better have been called "The Looping Problem".

Corollary: The language

$$\text{Loops} = \{M\#x \mid \text{loops}(M, x)\}$$

is not recursively-enumerable.

Proof: If Loops were recursively-enumerable, then the machine accepting it could be modified to get a machine accepting LoopsSelf, as follows: With input (an encoding of) M, a machine N would simply create M#M, then use this tape as an input for the machine accepting Loops. Thus N accepts LoopsSelf, which we already showed to be impossible.

Theorem 4: The language

$$\text{HaltsSelf} = \{M \mid \text{halts}(M, M)\}$$

is recursively-enumerable but not recursive.

Proof: HaltsSelf is accepted by a machine that, with input M, constructs M#M, then runs a universal TM wherein the rejecting state has been changed to accepting. Thus the modified machine will always accept or loop.

If HaltsSelf were recursive, then its complement would be by Observation 2. But the complement of HaltsSelf is LoopsSelf, which was shown to be not recursively-enumerable. Hence by Theorem 1, HaltsSelf is not recursive.

Corollary: The language

$$\text{Halts} = \{M\#x \mid \text{halts}(M, x)\}$$

is recursively-enumerable, but not recursive. (In Kozen, Halts is called HP, for “Halting Problem”.)

Proof: Halts is recursively-enumerable, because a universal TM can be modified to accept it by making every rejecting state an accepting state. If Halts were recursive, a machine accepting and always halting it could be modified to get a machine accepting HaltsSelf. With input M, that machine would create M#M and pass it to the machine Halts. But the Theorem showed that there is no such machine.

Corollary: The language

$$\text{Membership} = \{M\#x \mid \text{accept}(M, x)\}$$

Is recursively-enumerable, but not recursive. (In Kozen, Membership is called MP, for “Membership Problem”.)

Proof: Membership is recursively-enumerable, since it is exactly the set accepted by a universal TM. The complement of Membership is

$$\text{NonMembership} = \{M\#x \mid \neg\text{accept}(M, x)\}$$

It is easy to see that if NonMembership were recursively enumerable, so would NonMembership discussed earlier, but this is not the case.

Variants of the "Halting Problem"

Theorem 5: The language

$$\text{LoopsBlank} = \{M \mid \text{loops}(M, \varepsilon)\}$$

Is not recursively-enumerable. (Here ε represents the "blank" tape, i.e. the tape with no symbols.)

Caution: It is very important to notice that Theorem 5 cannot be dismissed as simply a "special case" of Loops being non-recursively-enumerable.

Proof: We show that if LoopsBlank were recursively-enumerable, then LoopsSelf would be also, but we showed the latter to be false.

Suppose N is a machine that accepts LoopsBlank. We will use N to get a machine N' that accepts LoopsSelf (which is impossible).

N' works as follows: Given input (an encoding of) machine M, N' creates an encoding of a new machine M' according to the following:

> M' appends string M to whatever is on the tape the value.
>
> (It is sometimes said that the ability to do so is "hard-wired into" or "programmed into" M', to help paint a mental picture.)

So if z were on the original tape of M', then zM would be on the tape after appending, and in particular, if ε were on the original tape, then M would be on the tape after insertion. Then M' behaves as if M from then on out. So in the particular case where ε is on the tape of M', the outcome will be the same as M with M on its tape.

N' then passes the description of M' to N, which was assumed to answer the question loops(M', ε). This answer is the result of N'. But the answer to loops(M', ε) is the same as the answer loops(M, M). So the net effect of N' is to accept LoopsSelf.

Corollary: The language

$$\text{HaltsBlank} = \{M \mid \text{halts}(M, \varepsilon)\}$$

Is recursively-enumerable, but not recursive.

Theorem 6: Let k be a fixed string

$$\text{Loops-k} = \{M \mid \text{loops}(M, k)\}$$

Is not recursively-enumerable.

Caution: The same caution applies as in Theorem 5.

Proof: The proof is similar to that of Theorem 5. However, in the present case, the constructed machine M' must *first erase* whatever is on the tape initially, *then* write M. So a machine that answers loops(M', k) can be used to answer for loops(M, M), which we've shown to be impossible.

Suppose that there is a machine N that accepts Loops-k. We use N to construct N' that accepts LoopsSelf, which is impossible.

N' with input M constructs M' that behaves as follows:

a. M' erases whatever is on its input tape.
b. M' writes the encoding M.
c. M' behaves as M on whatever is on the tape.

Thus M' will loop on input k iff M loops on input M. That is, machine N' accepts LoopsSelf.

Corollary: The language

$$\text{Halts-k} = \{M\#x \mid \text{halts}(M, k)\}$$

for a fixed string k, is recursively-enumerable, but not recursive.

Theorem 7 and 8: The languages

$$\text{LoopsSome} = \{M \mid \exists x \ \text{loops}(M, x)\}$$

$$\text{LoopsAll} = \{M \mid \forall x \ \text{loops}(M, x)\}$$

are not recursively-enumerable.

Proof: The same construction as in Theorem 6 is used. Since M' behaves the same on all input tapes, it loops on some tape iff M loops on itself. It also loops on all tapes iff M loops on itself.

We will eventually see that the complement of LoopsSome, called HaltsAll, is not recursively-enumerable either.

Theorem 9: The language

$$\text{HaltsSome} = \{M \mid \exists x \text{ halts}(M, x)\}$$

is recursively-enumerable, but not recursive.

Proof: This is not the same proof as in previous results, because we don't have just one tape to supply to a universal TM to do the test. Instead, we ostensibly need to enumerate the possible tapes and simulate M on each of them until one is found for which halts(M, x).

The enumeration of tapes has to be interleaved with the simulation, because the set of tapes is infinite, and thus we cannot enumerate them all "up front". Also, we cannot simulate them purely sequentially, as any individual simulation might loop. So we must arrange to generate, then simulate, M on an increasing set of tapes, incrementally, until one of the simulations halts, in which case M is accepted. This overall pattern is known as "***dovetailing***", and is seen as a very elaborate generalization of multiplexing. If none of the simulated computations halts, then the interleaving won't halt either. This is exactly the right thing, for we want the machine not to halt if $\neg\exists x$ halts(M, x).

To see that this language is not recursive, note that the complement is LoopsAll, as discussed in Theorem 8.

Caution: The next language to be discussed has a substantially different property than those discussed earlier: Neither it *nor* its complement are recursively-enumerable. The proof that shows this is trickier.

Theorem 10: The language

$$\text{HaltsAll} = \{M \mid \forall x \text{ halts}(M, x)\}$$

is *not* recursively-enumerable, nor is its complement (LoopsSome).

Proof: Suppose that HaltsAll were recursively-enumerable. Let N be a machine accepting HaltsAll. From this, we can create a machine N' that works as follows: With input M, construct a machine M' that, with input x, simulates M on M for x steps (interpreting x as a number). In other words, the simulating machine is like a universal machine with a specified limit on the number of steps, and that limit is set by the input string to M'.

Continuing the construction of M', if M halts on input M within x steps, then M' intentionally loops. If M on M does not halt within x steps, then M' halts.

Thus if loops(M, M) then M will not halt with input M for any number of steps, so M' will halt on all inputs. Also, if ¬loops(M, M) then M will halt on M within some number of steps. So M' will loop on some input.

In summary,

$$\text{loops}(M, M) \leftrightarrow \forall x \, \text{halts}(M', x)$$

Since N accepts M' iff halts(M', x), N' accepts M iff loops(M, M). But in Theorem 3, we showed that there is no such machine N'. Hence our supposition that HaltsAll is recursively-enumerable was wrong.

As mentioned, the complement of HaltsAll is LoopsSome, was already shown to be not recursively-enumerable.

Here is a related result, which looks similar to Theorem 10, but is actually a bit different.

Theorem 11: The language

$$\text{IsRecursive} = \{M \mid L(M) \text{ is recursive}\}$$

is *not* recursively-enumerable.

The reason that this is not identical to Theorem 10 is that in ∀x halts(M, x) could be false, but there still could be some other machine that totally accepts L(M), in which case L(M) would be recursive.

Proof: Suppose that IsRecursive is recursively-enumerable. Let N be a TM accepting IsRecursive. We show that we can then construct a machine N' that accepts LoopsSelf, which we know to impossible.

N' with input M constructs the description of a machine M' that behaves as follows:

> M' with input y saves y (e.g. by copying it onto a separate track of its tape) for later use. It then writes M#M on a different track and behaves as M with input M, as if a universal machine.
>
> If loops(M, M), then of course M' must loop, because that can't generally be detected.
>
> If halts(M, M), then M' runs a machine that accepts {y | halts(y, y)} on the original input y of M'.

The key property of the constructed machine M' is found in the *language* it accepts:

$$L(M') = \begin{cases} \{y \mid halts(y, y)\} & \text{if } halts(M, M) \\ \varnothing \text{ otherwise} \end{cases}$$

In the first case, we know L(M') is not recursive (although it is certainly r.e.), while in the second case L(M) = $\varnothing$ is recursive, as there certainly is an always-halting TM that accepts the empty language.

N' then passes M' to N, which by assumption accepts M' iff L(M') is recursive. But above we saw that L(M') is recursive iff ¬halts(M, M). Therefore

$$accepts(N', M) \leftrightarrow \neg halts(M, M)$$

so N' accepts LoopsSelf, which we showed to be impossible in Theorem 3.

The next result uses a proof similar to the preceding. It is highly useful in showing a variety of languages to be non-recursive. However, it must be used with caution, because it is strictly about properties of language and not the particular machine accepting the language.

Functional Properties

Consider the set RE of all TM encodings. We call the set RE because it defines the recursively-enumerable languages.

By a **functional property** of an element M in RE, we mean one that is true or false of the language L(M) independently of the particular machine used. In other words, if L(M) = L(M'), then M has the property iff M' does.

An example of a functional property is L(M) being *recursive*. The language is recursive, regardless of which of many machines accepts it.

An example of a non-functional property is "M has fewer than 100 states". For any M with this property, L(M) is accepted by some machine with the property and by some machines without it.

The reason that Theorems 10 and 11 are not equivalent is that Theorem 11 deals with a functional property, a language being recursive, whereas Theorem 10 deals with a non-functional property, a specific machine always halting.

Properties can be identified with **subsets** of RE for which the property is true.

A property is called **trivial** if it is identified only with the empty set $\varnothing$ or with all of Σ^*. In other words, it is true for no machines, or it is true for every machine.

Rice's Theorem

Let P be any non-trivial functional property identified with a subset of RE. Then the set $\{M \in \Sigma^* \mid P(M)\}$ is not recursive.

Put another way, the only recursive functional properties are the two trivial ones.

Proof: The proof is similar to the construction in the proof of Theorem 11. Suppose that P is a non-trivial functional property. We can assume that the empty set $\varnothing$ does *not* have property P, for if it did, work with the complementary property instead, which the empty set would not have. This assumption is used in an essential way later in the proof. It is not merely a convenience.

Let MP be the encoding of an arbitrary machine having property P.

Suppose that P is recursive, i.e. there is a machine N that can determine whether or not an arbitrary machine has property P. We will use N to construct a machine N' that can determine whether or not its input halts on itself, thereby obtaining a contradiction.

N' with input M constructs a machine M' which, with input y, sets aside y and simulates M on M. If halts(M, M), then N' runs MP on input y. If ¬halts(M, M), then M' must loop.

We can see that L(M') = L(MP) if halts(M, M) and L(M') = $\varnothing$ if ¬halts(M, M). So if we pass M' to N, to determine whether L(M') has property P, it effectively determines whether or not halts(M, M), which know is impossible.

Other examples of functional properties, to which Rice's Theorem thus applies, are:

- L(M) is context-free
- L(M) is regular
- L(M) is empty
- L(M) is Σ^*
- L(M) is finite
- L(M) is co-finite (its complement is finite)

Rice's theorem says that the listed properties are not recursive.

Functions vs. Languages

Turing machines can do more than just accept languages. They can compute *functions* of type $\Sigma^* \rightarrow \Delta^*$, where Σ is the input alphabet and Δ is a subset of the tape alphabet, called the *output alphabet*. Actually, because a machine doesn't necessarily halt, it is more accurate to say that Turing machines can compute *partial functions* of type $\Sigma^* \rightarrow \Delta^*$. For some $x \in \Sigma^*$, the machine might not halt, in which case the output is designated as $\perp$ for "undefined". It is important to realize that $\perp$ is not a value that is actually produced by the machine, but rather a symbol for divergent behavior.

A *recursive partial functions* is defined to be a computed by some Turing machine. A *recursive function* is a recursive partial function where the result is never $\perp$.

It is now clear that language acceptance is a special case of partial function computation. The machine can produce a standard value, such as the string '1' to indicate acceptance, and anything else for rejection. Divergence is one of the two forms of non-acceptance, as before.

The Source of "Recursively-Enumerable"

A function of the form $\Sigma^* \rightarrow \Delta^*$ can be thought of as *enumerating* a language. Think of the elements of as natural numbers. For example, if $\Sigma = \{0, 1\}$, one natural correspondence is:

Number	Element of {0, 1}
0	ε
1	0
2	1
3	00
4	01
5	10
6	11
7	000
. . .	. . .

Incidentally, this representation of the natural numbers is called the 2-adic representation. It is similar to binary, except that it has the property of being 1-1: there is a unique representation for each element.

The following theorem justifies the name "recursively-enumerable".

Theorem 12: A non-empty language is recursively-enumerable iff there is a recursive function f with the language as its range:

$$L = \{ f(x) \mid x \in \Sigma^* \}$$

Proof: Suppose that L is the range of a recursive function f. Then we can construct a TM that accepts L as follows:

Given an input y, successively generate f(0), f(1), f(2), . . . by simulating the turing machine with L as range. When and if an x is encountered such that y = f(x), accept y. If no such x is ever encountered, then this TM loops. Obviously y is in the range of f iff the machine accepts y.

Conversely, suppose L is a non-empty recursively-enumerable language. We need to exhibit a function f having y as its range.

If L is finite, then just define f(0) to be the first element of L, f(1) to be the second, and so on, cycling this way *ad infinitum*. Each element appears infinitely-many times in the enumeration, and the set is indeed the range of f.

If L is infinite, then let M be a turing machine accepting L. Multiplex simulations of M on an ever- larger subset of Σ^*, starting with ε, then 0, then 1, then 01, etc. The argument to f is one fewer than the goal for how many elements of L have to be generated. For f(0), only one element need be generated, and that element is f(0). The second element generated is f(1). And so on. Since L is infinite, we know that eventually f(i) is produced for any i. Multiplexing is necessary because the simulation on some elements of Σ^* may diverge, but we can't let the production of the next range element diverge.

Reducibility

An algorithmic problem P is said to be **reducible to** an algorithmic problem Q if a solution to Q can be used to construct a solution to P. However, we must be careful about what it means "to construct". Here's a way to make it more precise: using a *computable* function to map an instance of problem P into one of Q:

Suppose $L \subseteq \Sigma^*$ and $M \subseteq \Delta^*$ are languages. The problem of accepting L can be **reduced to** the problem of accepting M provided that there is a recursive function $f: \Sigma^* \rightarrow \Delta^*$ such that

$$L = \{ x \mid f(x) \in M \}$$

In other words, to determine whether $x \in L$, we compute f(x) and ask the question whether $f(x) \in M$. If this can be done, we write

$$L \leq_m M$$

The subscript m technically stands for “many-to-one reducibility”, meaning that there can in general be more than one element of L mapping to the same element of M under f.

The usual application of reducibility in computability uses the contrapositive:

$L \leq_m M$ establishes that if L is not recursively-enumerable, then neither is M,

or if L is not recursive, then neither is M,

as in “the self-looping problem reduces to the blank-tape looping problem”. In this case, what is the corresponding recursive function? It is the construction that we defined which took an arbitrary machine M to a machine M’, such that M’ loops on the blank tape if M loops on M.

We can recast some of the proofs done earlier as reductions. Some mappings are obvious and others more subtle:

$HaltsSelf \leq_m Halts$ (obvious)

$NonMemberSelf \leq_m NonMemberSelf$ (obvious)

$LoopsSelf \leq_m LoopsBlank$

$LoopsSelf \leq_m Loops\text{-}k$

$LoopsSelf \leq_m LoopsSome$

$LoopsSelf \leq_m LoopsAll$

The Recursion Theorem

The recursion theorem uses the idea of true self-reference, specifically, a TM can write on its tape its *own description* in the course of computation, which it can subsequently use. In other words, machine M can, with adequate pre-planning in its construction, make use of the encoding of M as if it were a constant.

It takes some contriving to accomplish this, and we’ll leave it up to the reader for now.

An example of applying this form of self reference is a proof of the unsolvability of the halting problem that doesn’t use Diagonalization.

Theorem 13 (Halting Problem re-done):

The function h defined by h(M) = 1 if halts(M, M), 0 otherwise, is not recursive.

Proof: Suppose that h is recursive. Then so would the function g computed by machine G be recursive:

G with input M computes h(G) (M is ignored).

> If h(G) = 1, then G intentionally loops.
> If h(G) = 0, then G returns 1.

Thus G with input G loops if halts(G, G), which is contradictory.
Also, G with input G returns 1 if ¬halts(G, G), which is also contradictory. So in either case, halts(G, G) or not, we have a contradiction.

Another application of self-reference follows.

Theorem 14: The language MinLength =

> {M | M is a minimal-length encoding of a machine accepting L(M)}

is not recursively-enumerable.

Proof: Suppose that MinLength is recursively-enumerable. Let E be an always-halting machine that enumerates MinLength, in that E run on any input produces a minimal length encoding of some machine.

Construct a machine C that runs as follows: With input w:

- Obtain the encoding C.
- Run E on successively-larger inputs {0, 1, 2, 3, . . .} some machine D is produced with |D| > |C|.
- Run D on the original input w.

By assumption on E, |D| is a minimal encoding of a machine accepting whatever D accepts. But C produces the same result that D does, and by the criterion, C has a shorter encoding that D.

So D couldn't be minimal after all. Therefore E cannot exist, and MinLength is not recursively-enumerable.

The practical impact of Theorem 14 is that there is no way to determine whether a given program for an acceptor is actually the minimal-length one.

The idea of self-reference is captured formally in the following.

The Recursion Theorem (Kleene)

Let f: $\Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ be any partial function computable by a Turing machine.

Then there is a Turing machine R that computes the function r defined by:

$$\forall x \in \Sigma^* \qquad r(x) = f(R, x)$$

where R as an argument means the encoding of R.

We will not prove the recursion theorem here. We just wanted to show how to express the use of a machine's own encoding in a computation.

From a demonstration viewpoint, the access by a program of the program's own encoding can be shown by programs that print their own code, sometimes called "Quines" after the logician Willard Quine. Here is an example of a C program that does this, as offered in Kozen:

```
char *s="char *s=%c%s%c;%cmain(){printf(s,34,s,34,10,10);}%c";
main(){printf(s,34,s,34,10,10);}
```

If you compile and run this on a computer that has a C-compiler (the Unix commands for which would be 'cc self.c; ./a.out', where this program has been named self.c), you should get output which is identical to the program itself.

The recursion theorem indicates that this idea can be taken a step further, allowing the code to be used within the program, along with code for doing other stuff, which is still a part of the overall code. We'll leave it to the reader to devise a program that exactly demonstrates this capability, by showing how to encode r above for any given function f.

Recursive Enumerability and Logic

The formulas in a logical theory are a good example of a set that is typically recursively-enumerable. Assume that there is a finite (or perhaps only recursive) set of axioms as starting points, theorems can be generated by applying the rules of inference.

If a sentence is a theorem, it will eventually be generated. So a Turing machine, for example, could do the generation.

It is less clear that the set of theorems is recursive. There is no evident way to determine that a sentence (closed formula) is *not* a theorem. Indeed, the set of theorems often is *not* recursive. When a theory has the property that its set of theorems is recursive, the theory is called **decidable**.

Recall that a theory is **complete** provided that, for any sentence, either the sentence or its negation is provable. Since any sentence must be true or false with respect to a set of axioms, if a theory is incomplete, there will be sentences that are not theorems but which are nonetheless true.

Theorem 15: If a theory is complete, then it is decidable.

Proof: For the reader.

Corollary: If a theory is undecidable, then it is incomplete.

Incompleteness of Peano Arithmetic

The theory N of Peano Arithmetic was introduced in our discussion of logic. The most celebrated result of Kurt Gödel is that, provided N is consistent (which is not really in doubt) it is incomplete. Moreover, any consistent extension of N is likewise incomplete.

Kozen offers a proof of this incompleteness which is different from Gödel's, in addition to a sketch of Gödel's. The first proof uses the idea of Turing machines.

Kozen uses the notation Th(N) to represent the sentences that are true with respect to theory N (i.e. sentences φ such that N |= φ). This notation does not seem so well-chosen, since it could also be confused with the *theorems* of N (i.e. sentences φ such that N |– φ).

Theorem 16: The true sentences of N are not recursively-enumerable.

Kozen shows this result by reducing the halting problem to the determination of whether a formula is true in N.

Here is an outline of the proof. We have shown that the set

$$\text{Loops} = \{ M\#x \mid \neg \text{halts}(M, x) \}$$

is not recursively-enumerable. We reduce this set to the set of true sentences in N by constructing, for a given M and x, a formula γ such that

$$\neg \text{halts}(M, x) \text{ iff } N \models \gamma$$

This is done by representing, within the language of N, a numeric predicate $\text{VALCOMP}_{M, x}$ such that

$$\text{VALCOMP}_{M, x}(v)$$

where v is an encoded string, is a *valid computation history* of M started on x. Thus M halts on x iff there is such a history, so the formula γ is

$$\exists v \ \text{VALCOMP}_{M, x}(v)$$

Consult the text for the details of VALCOMP, which involve ways to construct and decompose strings using numeric encodings and the operations of N.