CSCE 551 — Notes on Self-Reference in Computation

Stephen A. Fenner

March 29, 2024

Abstract

Computing machines can refer to themselves while performing other duties. Some of the material in these notes is also covered in Sipser [Sip13, Section 6.1] in a different way.

1 The Recursion Theorem and Its Proof

When we proved the undecidability of A_{TM}, the acceptance problem for Turing machines (TMs), we set things up so that a TM F could ask an assumed decider for A_{TM}, "Am I going to accept my own description?" Whichever way the decider answers, F does the opposite, proving D incorrect.

The fact that a machine can refer to itself has much wider applications than just this proof. Here we take the idea of self-reference and run with it. We'll see how self-reference can be baked into a TM performing any computational task, as if the TM has its own description written somewhere on its tape "for free." This ability results from the following theorem, first proved in a different form by Stephen Kleene in the 1930s [Kle38]:

Theorem 1.1 (Recursion Theorem)**.** *For any TM A, there exists a TM R that on any input w behaves like A on input $\langle R, w\rangle$.*

Here, A represents an arbitrary computational task to be performed. It takes as input a pair whose first component is meant to be a string describing a TM and whose second is an arbitrary string w. R on input w then performs A, but the first part of A's input is the description of R itself. In this way, R gets to "see" itself while performing task A. (The phrase, "behaves like" in the above theorem means "has the same *eventual* behavior," i.e., accepting, rejecting, looping, or outputting something, whichever is appropriate.)

The construction of R in the following proof of Theorem 1.1 uses a trick similar to that used in the proof that A_{TM} is undecidable: a machine running on its own description as input.

Proof of Theorem 1.1. Given any fixed TM A, let

$F :=$ "On input $\langle M\rangle$, where M is a TM:

1. Let

   $R :=$ 'On input w:

   (a) Run M on input $\langle M\rangle$ and (assuming it halts) let r be its output.

   (b) Run A on input $\langle r, w\rangle$.'

2. Output $\langle R \rangle$."

F is clearly a transducer. Given any TM M as input, F halts and outputs a string describing a TM. Now let R be the TM whose description is output by F on input $\langle F \rangle$. Then by substituting $\langle F \rangle$ for $\langle M \rangle$ as the input to F, we see that R behaves as follows:

$R =$ "On input w:

1. Run F on input $\langle F \rangle$ and (if it halts) let r be its output.

2. Run A on input $\langle r, w \rangle$."

F halts on input $\langle F \rangle$ in step 1, and by definition its output is $\langle R \rangle$ itself! Thus $r = \langle R \rangle$ in step 1, and so R simulates A on input $\langle R, w \rangle$ in step 2. □

Variants of the Recursion Theorem were proven by other people, including Hartley Rogers (see [Rog67]). Many people find this result somewhat perplexing and at least a bit paradoxical. It refutes an early, naive criticism of strong AI along the lines of, "Machines can't act like humans because humans are self-aware and machines aren't." Well, machines actually *can* be made self-aware, by the theorem just proved. The first few pages of Chapter 6 of Sipser [Sip13] also give a nice description of why this result is viewed as a paradox.

There is nothing special about the Turing machine model of computation. Theorem 1.1 applies to any model of general-purpose computation, including abstract mathematical models but also general-purpose programming languages like C, C++, Java, Python, Lisp, ... More on that in some exercises below.

2 How to Use the Recursion Theorem

The Recursion Theorem says that when writing an algorithm implemented by a TM R, it is perfectly legal to refer to $\langle R \rangle$ itself inside the algorithm, e.g., $R :=$ "On input w: $\ldots \langle R \rangle \ldots$." It is as though R has its own description $\langle R \rangle$ sitting somewhere on its tape, with which it can do whatever it likes. To illustrate, here is a quick proof that A_{TM} is undecidable:

> Let D be an arbitrary decider. Let
>
> $R :=$ "On input w:
>
> 1. Run D on input $\langle R, w \rangle$.
> 2. If D accepts then reject; else accept."
>
> For every w, D accepts $\langle R, w \rangle$ iff R does not accept w. Hence D cannot decide A_{TM}.

Here we refer to R itself in step 1 of R's algorithm. Theorem 1.1 says there is no problem at all with this. To unpack R a bit, the implicit task A being performed is

$A :=$ "On input $\langle r, w \rangle$:

1. Run D on input $\langle r, w \rangle$.

2. If D accepts then reject; else accept."

The Recursion Theorem says that we can get a TM R by substituting $\langle R \rangle$ for r inside A.

Generally, we can pretend that R starts with its own description on its its tape, but that is really a simplification. No TM starts with its own description on its tape. What really happens is that R spends the initial part of its computation *obtaining* its own description for its tape (by running the TM F on input $\langle F \rangle$, where F is the TM defined in the proof of Theorem 1.1).

2.1 Some Positive Results

Here is a self-reproducing TM R:

$R :=$ "On input w:

1. Output $\langle R \rangle$."

R ignores its input and outputs its own description. The corresponding task A is

$A :=$ "On input $\langle r, w \rangle$:

1. Output r."

Exercise 2.1. Here is a classic programming exercise that is highly rewarding once you see how to do it: In your favorite general-purpose programming language, write a program that reads no input but produces output identical to its own source code. The recursion theorem says that such self-reproducing programs always exist, and there are usually several ways to write them. There is a self-reproducing C program that is 80 characters long[1]. There are self-reproducing C programs that are well-formatted and easily readable with comments.

[Note: You can find several of these programs online. If you copy one and submit it along with a clear citation of its source, then there is no honor code violation, but you will be denying yourself the edifying and revelatory joy of discovering one on your own.] □

In the same vein, here is a self-recognizing decider, that is, a TM R that accepts its own description and rejects everything else, i.e., $L(R) = \{\langle R \rangle\}$:

$R :=$ "On input w:

1. If $w = \langle R \rangle$ then accept; else reject."

Exercise 2.2. What is the task A corresponding to the TM R above? □

Exercise 2.3. In your favorite general-purpose programming language, write a program that reads a text file (from standard input, maybe), and if the input exactly matches the source code of the program, outputs "That's me!" and quits; otherwise, it outputs "Nope, not me" and quits. □

If you don't care about efficiency, the Recursion Theorem gives you a way to make your high-level algorithm recursive in the traditional programming sense of the word. Here is a TM that (inefficiently) computes the factorial function (we forgo the angle brackets this time):

$R :=$ "On input a natural number n:

1. If $n = 0$, then output 1.

[1] assuming the `printf` function is available without an `#include` directive

2. Else,

   (a) Run R on input $n-1$ and let p be its output.

   (b) Output pn."

Exercise 2.4. What is the task A corresponding to the factorial-computing R above? □

Exercise 2.5. What do you think the following TM R does? (Assume that R was constructed in the manner of the proof of Theorem 1.1.)

$R :=$ "On input w:

1. Run R on input w."

Justify your answer. □

If you're puzzled by that last exercise, try the next one first, then come back to the previous one.

Exercise 2.6. What does the following TM R do (i.e., accept, reject, or loop) on input ε?

$R :=$ "On input w:

1. If $w \neq \varepsilon$, then reject.

2. Run R on input ε.

   - If R accepts ε, then reject.
   - If R rejects ε, then accept."

Prove your answer. □

2.2 Some Negative Results

Besides showing that A_{TM} is undecidable, the Recursion Theorem can be used to show a variety of other undecidability/Turing-nonrecognizability/uncomputability results. The next theorem is a strengthening of one found in Sipser [Sip13, Section 6.1] but with essentially the same proof.

Definition 2.7. We say that a TM is *minimal* if there is no equivalent TM with a shorter description. That is, TM M is minimal if $L(M) \neq L(N)$ for every TM N such that $|\langle N\rangle| < |\langle M\rangle|$. We let

$$MIN_{\mathsf{TM}} := \{\langle M\rangle : M \text{ is a minimal TM}\} .$$

Clearly, every Turing-recognizable language has at least one minimal TM recognizing it. Thus MIN_{TM} is an infinite language.

Theorem 2.8. *MIN_{TM} has no infinite Turing-recognizable subsets.*[2]

Corollary 2.9. *MIN_{TM} is not Turing-recognizable.*

[2] In the literature, an infinite language that has no infinite Turing-recognizable subsets is called *immune*. Clearly, an immune language cannot be Turing-recognizable (being a subset of itself).

Proof of Theorem 2.8. Suppose there exists some infinite Turing-recognizable language $S \subseteq MIN_{\mathsf{TM}}$. Every string in S encodes a minimal TM. Fix an enumerator E enumerating S. Let R be the following TM:

$R :=$ "On input w:

1. Run E until it prints a string $\langle M \rangle$ such that $|\langle R \rangle| < |\langle M \rangle|$.
2. Run M on input w."

Since S is infinite, E will print arbitrarily long strings, and so step (1) will eventually terminate, producing a minimal TM M. Then, since R simulates M on every input string, we have $L(R) = L(M)$. But $|\langle R \rangle| < |\langle M \rangle|$, contradicting the fact that M is minimal. Hence no such S can exist. □

A corollary to Theorem 2.8 says that you can't computably optimize a program with respect to its size. (For any function $f : \Sigma^* \to \Sigma^*$, we define the *range* of f as $\text{range}(f) := \{f(w) \mid w \in \Sigma^*\}$.)

Corollary 2.10. *Let f be any function that, given a TM M as input, outputs a minimal TM equivalent to M. No such f can be computable.*

Proof. Suppose there there exists such a computable f. The range of any computable function is Turing-recognizable (we easily prove this as Proposition 2.11, below). Thus $\text{range}(f)$ is an infinite Turing-recognizable subset of MIN_{TM}, which does not exist by Theorem 2.8. □

I sometimes prove this proposition in class:

Proposition 2.11. *Let f be any computable function. Then* $\text{range}(f)$ *is Turing-recognizable.*

Proof. Consider the following enumerator E:
$E :=$ "On no input:

1. Cycling through all strings w:
   (a) Compute $x := f(w)$
   (b) Print x"

E evidently enumerates $\text{range}(f)$. Thus $\text{range}(f)$ is enumerable, hence Turing-recognizable by a theorem proven in class. □

Exercise 2.12. Let $ALMOST\text{-}MIN_{\mathsf{TM}}$ be the language

$$ALMOST\text{-}MIN_{\mathsf{TM}} := \{\langle M \rangle : M \text{ is a TM, and any equivalent TM } N \text{ satisfies } |\langle N \rangle^2| \geq |\langle M \rangle|\} .$$

(So elements of $ALMOST\text{-}MIN_{\mathsf{TM}}$ are the TMs whose size is no more than the square of their minimal equivalent TMs.) Clearly, $MIN_{\mathsf{TM}} \subseteq ALMOST\text{-}MIN_{\mathsf{TM}}$, and so $ALMOST\text{-}MIN_{\mathsf{TM}}$ is infinite.

1. Show that $ALMOST\text{-}MIN_{\mathsf{TM}}$ has no infinite Turing-recognizable subsets.
2. Show as a corollary that no computable program-size optimizer can get within the square of the minimum size of a TM equivalent to its input TM.

3. There is nothing magic about the squaring function above. How can you generalize this result?

□

The next proposition says that, even if you know that a TM decides a finite language L, you cannot compute an upper bound on the number of strings in L.

Proposition 2.13. *Let $f : \mathbb{N} \to \mathbb{N}$ be a function such that, for every decider M such that $L(M)$ is finite, $f(\langle M \rangle)$ is an upper bound on the cardinality of $L(M)$. No such f is computable.*

Proof. Suppose there is such a computable f. Let

$R :=$ "On input w:

1. Let $n := f(\langle R \rangle)$.
2. If $|w| \leq n$ then accept; else reject."

R is evidently a decider, and $L(R)$ only contains strings of length $\leq n$, and so $L(R)$ is finite. But there are at least $n + 1$ many strings of length $\leq n$ (including ε), which contradicts the fact that $n = f(\langle R \rangle)$ is an upper bound on the cardinality of $L(R)$. □

With a little bit of clocking, we can actually get a much stronger result:

Theorem 2.14. *There is no TM S such that, for every decider M such that $L(M)$ is finite, S on input $\langle M \rangle$ halts and outputs an upper bound on the cardinality of $L(M)$.*

What makes Theorem 2.14 stronger than Proposition 2.13 is that in Proposition 2.13 we assume that the machine computing the upper bound halts on *all* inputs, but we don't make that assumption in Theorem 2.14. (S could loop if M is not a decider or if $L(M)$ is infinite.)

Proof of Theorem 2.14. Assume such an S exists. Let

$R :=$ "On input w:

1. Run S on input $\langle R \rangle$ for $|w|$ many steps.
2. If S has not halted on $\langle R \rangle$ within $|w|$ steps, then reject.
3. Otherwise, let t be the number of steps it takes for S to halt, and let n be the output of S, where n is a natural number. (If S's output does not encode a natural number, then reject.)
4. If $|w| \leq t + n$, then accept; else reject."

R is evidently a decider. Furthermore, $L(R)$ is finite, which can be seen as follows: If S loops on input $\langle R \rangle$, then R rejects in step (2) regardless of its input w, and thus $L(R) = \emptyset$, which is finite; on the other hand, if S halts on input $\langle R \rangle$, then neither the time t it takes to halt nor its output n depend on w, which means that R will only accept strings of length $\leq t + n$, and so $L(R)$ is finite in this case also.

Since R decides a finite language, S must halt on input $\langle R \rangle$ by assumption. Letting t be the number of steps it takes for S to halt on input $\langle R \rangle$ and letting n be S's output, we see that R accepts all strings w such that $t \leq |w| \leq t + n$. There are at least $n + 1$ many such strings, which contradicts the assumption that S outputs an upper bound on the cardinality of $L(R)$. Thus no such S can exist. □

The next theorem, in the same spirit as the previous theorem, says that even if we know that a TM recognizes a decidable language, we cannot computably find a decider for that language. As an exercise, you will fill in some of the details of the proof.

Theorem 2.15. *There is no TM S such that, for any TM M such that $L(M)$ is decidable, S on input $\langle M \rangle$ outputs (a string describing) a decider for $L(M)$.*

Proof sketch. Suppose such an S exists. Let

$R :=$ "On input w:

1. Run S on input $\langle R \rangle$, and (if it halts) let d be its output.
2. If d does not encode a TM, then reject.
3. Otherwise, let $d = \langle D \rangle$ for some TM D.
4. Sequentially, for each string x such that $|x| < |w|$,
   (a) Run D on input x (with no time limit), ignoring the result, if any.
5. Run D on input w:
   (a) If D accepts w, then reject.
   (b) If D rejects w, then accept.
   (c) (Else, loop.)"

Then $L(R)$ is decidable, but S on input $\langle R \rangle$ does not output a decider for $L(R)$. (See Exercise 2.16 for details.) Contradiction. Thus no such S exists. □

Exercise 2.16. Fill in the details in the proof of Theorem 2.15 above. Here is a guide:

1. First, explain why $L(R)$ is decidable, regardless of the behavior of S or D (or whether or not the latter even exists). (R may loop, but $L(R)$ is still decid-*able*.) The mysterious Step 4, or something like it, is actually crucial to make this argument.
2. Having established (1.), now use the assumption about S to argue that S on input $\langle R \rangle$ outputs some decider D.
3. Show that D does not decide $L(R)$, which contradicts the assumption made about S. (Use what happens in Step 5 to argue this.)

□

2.3 Incompleteness of Mathematical Theories

In 1931, Kurt Gödel shocked the mathematical community by showing that any formal mathematical system that satisfies a few minimal properties must include statements that are neither provable nor refutable in the system (that is, neither the statement nor its negation is provable in the system) [Göd31]. (For an English translation, see [Göd86].) Such statements are called *formally undecidable*. Perhaps sensing how profoundly disturbing his result would be, he gave a long, careful, minutely detailed proof of it. However, equipped with the Recursion Theorem and our knowledge and intuitions about programming, we can give a high-level proof of Gödel's result very quickly:

Proof sketch. Fix a formal mathematical system Π. (Commonly used systems are first-order Peano arithmetic and Zermelo-Fraenkel set theory.) Let R be the following TM:

$R :=$ "On any input:

1. Cycling through all strings p,
   (a) If p encodes a proof in Π of the statement, 'R loops on input ε,' then accept.
   (b) Else, go on to the next p."

So on input ε, R searches for a proof that it will loop on input ε. If it ever finds such a proof, it halts and accepts. If it does not find a proof, it will of course run forever (loop). Let ψ be the statement, "R loops on input ε." Then if there is a proof of ψ in the system Π, then ψ is *false*, because R will eventually find such a proof and halt. On the other hand, if there is *no* proof of ψ in the system Π, then ψ is clearly *true*.

So either Π proves a false statement about strings or it is unable to prove a true statement about strings. The former means that Π is *unsound*, and thus not a system that anyone would want to use. Assuming that Π is sound, then, we have no alternative but to conclude that Π cannot prove the true statement ψ, and being sound, Π cannot prove $\neg\psi$ either. Thus ψ is formally undecidable by Π, assuming Π is sound.[3] □

In the proof above, we implicitly make some other basic assumptions about Π besides its soundness:

- The language of Π must be expressive enough to render ψ as a formal sentence.
- Checking proofs in Π must be algorithmic, that is, given any string p and formula φ in the language of Π, there is an algorithm that decides whether p encodes a correct proof of φ.

Both first-order Peano arithmetic and Zermelo-Fraenkel set theory have these two properties (the proofs are routine and not deep), and it is almost universally believed (but unprovable!) that both theories are sound.

3 Extensions to the Recursion Theorem

In this section we give some extensions and generalizations to Theorem 1.1. For convenience, we will drop angle brackets around TMs, identifying them with the strings that describe them. It will be clear from the context whether we are referring to a TM or to its string. We will also assume that every string encodes some TM; if the string is not a syntactically correct TM description, we will take it to encode a trivial TM that loops on all inputs.

Theorem 1.1 has the form, "For any TM A, there exists a TM R that ...." The next theorem is based on the simple observation that passing from A to R can be done algorithmically (i.e., "effectively").

Theorem 3.1 (Recursion Theorem (Effective Version))**.** *There is a transducer G that on any input TM A outputs a TM R that, given input w, behaves like A on input $\langle R, w\rangle$.*

[3] What we are calling *sound* is often referred to as *ω-consistent* in the literature.

Proof. Let

$G :=$ "On input TM A:

1. Let

   $F :=$ 'On input TM M:

   (a) Let

   $R :=$ "On input w:

   i. Run M on input M and (assuming it halts) let r be its output.

   ii. Run A on input $\langle r, w \rangle$."

   (b) Output R.'

2. Run F on input F (and output the result)."

The rest of the proof is similar to that of Theorem 1.1. □

The next theorem is equivalent to the Recursion Theorem. It was proved some years later by Hartley Rogers [Rog67] and is known as Rogers's Fixed Point Theorem.

Theorem 3.2 (Fixed Point Theorem)**.** *For every computable function f there exists a TM R that behaves like the TM $f(R)$. (R is a "fixed point" of f.)*

Proof. Given f, we can invoke Theorem 1.1 to define

$R :=$ "On input w:

1. Let $M := f(R)$.

2. Run M on input w."

Evidently, R and $f(R)$ behave the same way on all inputs w. □

The Recursion Theorem also follows easily from the Fixed Point Theorem. The two theorems are so similar that some authors confuse the two and call the Fixed Point Theorem the Recursion Theorem.

Proof of Theorem 1.1 using Theorem 3.2. Given TM A, define

$f :=$ "On input r:

1. Let

   $M :=$ 'On input w:

   (a) Run A on input $\langle r, w \rangle$.'

2. Output M.

Evidently, f is a computable function. By Theorem 3.2, there exists a TM R such that R behaves like TM $f(R)$. Letting $M := f(R)$, we see that M implements the following:

"On input w:

1. Run A on input $\langle R, w\rangle$."

But R behaves like M, so R also implements the above, and hence satisfies Theorem 1.1. □

The Fixed Point Theorem can also be made effective, just like the Recursion Theorem. We can also make f and the fixed point R depend (computably) on an extra parameter y. This more general theorem can be useful in many ways. In the theorem below, y is the extra parameter that f takes as input, and the corresponding fixed point is now a computable function $n(y)$ of the parameter y.

Theorem 3.3 (Fixed Point Theorem With Parameter)**.** *For every computable function f there exists a computable function n such that, for all strings y, TM $f(\langle n(y), y\rangle)$ behaves like TM $n(y)$.*

Proof. Let G be the transducer defined in the proof of Theorem 3.1. Given a computable function f, define the computable function n as follows:

$n :=$ "On input y:

1. Let

   $A :=$ 'On input $\langle r, w\rangle$, where r and w are strings:

   (a) Compute $R := f(\langle r, y\rangle)$.

   (b) Run R on input w.'

2. Run G on input A."

Since G is a transducer (and thus halts with output on all inputs), n is a computable function. Now fix any string y, and let TM $R := n(y)$. Then $R = G(A)$, where TM A on any input $\langle r, w\rangle$ behaves like the TM $f(\langle r, y\rangle)$ on input w. By the properties of G, we then have that R on input w behaves like A on input $\langle R, w\rangle$, which in turn behaves like $f(\langle R, y\rangle)$ on input w. But $R = n(y)$, meaning that TM $n(y)$ behaves like TM $f(\langle n(y), y\rangle)$ on any input w, which is what we were to prove. □

The following theorem is a recasting of a theorem of Raymond Smullyan [Smu61] (see also [Smu93] for this and a host of other variants). It allows for mutual recursion between two algorithms.

Theorem 3.4 (Double Recursion Theorem)**.** *Let A and B be TMs. There exist TMs R and S such that, on any input w, R behaves like A on input $\langle R, S, w\rangle$ and S behaves like B on input $\langle R, S, w\rangle$.*

In other words, two TMs can refer to themselves and to *each other* while performing different tasks. The proof is a variation on that of Theorem 1.1.

Proof of Theorem 3.4. Let TMs A and B be given. Define

$E :=$ "On input $\langle M, N\rangle$:

1. Let

   $R :=$ 'On input w:

   (a) Run M on input $\langle M, N\rangle$ and (if it halts) let r be its output.

(b) Run N on input $\langle M, N\rangle$ and (if it halts) let s be its output.

(c) Run A on input $\langle r, s, w\rangle$.'

2. Output R."

and define

$F :=$ "On input $\langle M, N\rangle$:

1. Let

   $S :=$ 'On input w:

   (a) Run TM M on input $\langle M, N\rangle$ and (if it halts) let r be its output.

   (b) Run TM N on input $\langle M, N\rangle$ and (if it halts) let s be its output.

   (c) Run B on input $\langle r, s, w\rangle$.'

2. Output S."

E and F are both transducers (that output TM descriptions).

- Let R be the output of E on input $\langle E, F\rangle$;
- let S be the output of F on input $\langle E, F\rangle$.

Then in the algorithms for both E and F, we have $r = R$ and $s = S$. Therefore, for any input w, R simulates A on input $\langle R, S, w\rangle$ and S behaves like B on input $\langle R, S, w\rangle$. □

Exercise 3.5. In your favorite general-purpose programming language, write two programs, `prog1` and `prog2`, so that

- `prog1` takes no input but outputs text identical to the source code of `prog2`, and
- `prog2` reads text from standard input and, if the text exactly matches the source code of `prog1`, outputs "`That's prog1`" and quits; otherwise, it outputs "`Not prog1`" and quits. (`prog2` is not allowed to directly access the source file of `prog1`.)

□

Here is a fixed-point version of the Double Recursion Theorem.

Theorem 3.6. *For any two computable functions f and g, there exist TMs R and S such that R behaves like TM $f(R, S)$ and S behaves like TM $g(R, S)$.*

Proof. See Exercise 3.7, below. □

Exercise 3.7. Prove Theorem 3.6. □

Exercise 3.8. Formulate and prove a Triple Recursion Theorem. □

References

[Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der *Principia Mathematica* und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. doi:10.1007/BF01700692.

[Göd86] K. Gödel. On formally undecidable propositions of *Principia Mathematica* and related systems I. In S. Feferman, editor, *Kurt Gödel. Collected Works*, volume I, pages 145–195. Oxford University Press, 1986.

[Kle38] Stephen C. Kleene. On notation for ordinal numbers. *Journal of Symbolic Logic*, 3(4):150–155, 1938. doi:10.2307/2267778.

[Rog67] Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967. Reprinted by MIT Press, 1987. ISBN 9780262680523.

[Sip13] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2013.

[Smu61] R. Smullyan. Theory of formal systems. *Annals of Mathematics Studies No. 47*, 1961.

[Smu93] Raymond M. Smullyan. Symmetric and Double Recursion Theorems. In *Recursion Theory for Metamathematics*. Oxford University Press, 07 1993.