Kleene's Second Recursion Theorem and Self-Referencing Programs

Last Updated March 25th, 2024

Kleene's Second Recursion Theorem

"Know Thyself"

Socrates

Consider a computable function $f(x, y)$, where x is viewed as a Gödel number of some program and y is some other input. The following are some statements that could be made in an informal program that computes f.

- Print the instructions of P_x.
- Simulate the computation of P_x on input y.
- Count the number of Jump instructions that are executed in the computation of P_x on input y.
- Send program x and input y to another computer in the network.
- Return, as a single natural-number encoding, the tuple of configurations that constitutes the computation of P_x on input y.

Now suppose we take f's program statements and re-write them in a self-referencing way, to where we get statements like the following ones.

- Print *my* instructions.
- Simulate *myself* on input y.
- Count the number of Jump instructions that I execute when I'm computing input y.
- Send *myself* and y to another computer in the network.
- Return, as a single natural-number encoding, the tuple of configurations that constitutes *my* computation on input y.

A program that makes one or more references to its own Gödel number is said to be **self-referencing** (or **self-knowing**). Note that this is *not* the same as a *recursive program* that makes one or more calls to itself using smaller-sized inputs.

Catch-22 for a self-referencing program P

1. For P to know its Gödel number, it must know each of its instructions.
2. Some instructions, such as "print myself", requires P to know its Gödel number.

Proposed Solution to Catch-22

1. Assume for the sake of argument that, after replacing statements about x with statements about itself, that there does in fact exist a program P_e with Gödel number e that computes the resulting function.
2. Then P_e is a function of the single variable y (since variable x has been assigned constant e).
3. Therefore, we have, for all y,
$$\phi_e(y) = f(e, y).$$
In other words, there is a program P_e that, on input y computes $f(e, y)$, and thus makes references (to e which been substituted for x) to its own Gödel number.
4. Thus, we have reduced the problem to that of finding a Gödel number e that satisfies the above equation.
5. Stephen Kleene's second recursion theorem states that such an e does exist!

Kleene's Second Recursion Theorem. Let $f(x, y)$ be a computable function that takes as input a Gödel number x, and some additional input y. Then there is a Gödel number e for which $\phi_e(y) = f(e, y)$.

Example 1. Consider the URM computable function $f(x, y)$ which, on inputs x and y, simulates the computation $P_x(y)$, and returns the number of times that a jump instruction is executed during the computation $P_x(y)$. Then by the 2nd recursion theorem, there is a program P_e for which $P_e(y) = f(e, y)$, and so, for input y, P_e computes the number of times that its own self executes a jump instruction during its computation with input y.

Suppose $\hat{P}$ computes $f(x, y)$, meaning $\hat{P}(x, y) = f(x, y)$ for all inputs x and y.

Interviewer: "What do you do for a living $\hat{P}$?"

$\hat{P}$**:** "I simulate program P_x on input y and output the number of jump instructions that were executed by P_x during the simulation.

Now let e be a Gödel number for which $P_e(y) = f(e, y)$.

Interviewer: "What do you do for a living P_e?"

P_e**:** "I simulate myself on input y and output the number of jump instructions that my simulated self made during the simulation.

$\square$

Proof of Kleene's Second Recursion Theorem. The idea behind the proof is to divide the construction of the desired program $P = ABC$ into three parts: A, B, and C which we now describe. Assume that y is the input to P.

Part A.
- Move y to register R_2.
- Place B's Gödel number b in R_1.

Part B.
- Use b in R_1 to compute A's Gödel number a.
- Compute C's Gödel number c.
- Compute
$$e = \gamma(\gamma^{-1}(a), \gamma^{-1}(b)\gamma^{-1}(c)) = \gamma(ABC) = \gamma(P),$$
the Gödel number of the concatenation of A's, B's, and C's instructions.
- Place e in R_1, with y remaining in R_2.

Part C. Compute $f(e, y)$.

Notes.

1. The most straightforward of the three is part C, since its sole purpose is to compute function f which is assumed URM computable, and so C's instructions consist of the instructions of the URM program used to compute f.

2. The clever part of the above program is understanding how A is able to compute B's Gödel number and vice versa. This is actually made possible by an elementary use of the s-m-n theorem.

3. Consider the function $g(x, y) = x$. By the s-m-n theorem, there is a total computable function $k(x)$ for which
$$\phi_{k(x)}(y) = g(x, y) = x.$$
Thus, $\phi_{k(x)}(y)$ is a constant function which, for any input y, always outputs x (in register R_1).

4. Then define A's Gödel number to be equal to $k(b)$. This works because, on input y, program A outputs
$$\phi_{k(b)}(y) = b$$
in register R_1, and has the side effect of placing y in R_2 via an initial $T(1, 2)$ statement. Therefore A works in exactly the way it was described in Part A above.

Given that $a = k(b)$ we may now describe B's program as follows.

Program B

Input Gödel number z.

Compute Gödel number $k(z)$.

Compute $c = \gamma(C)$.

Return

$$\gamma(\gamma^{-1}(k(z)), \gamma^{-1}(z), c).$$

Important: notice that B's program does *not* depend on knowing A's Gödel number a. If it did, then it would create a circularity error, since $a = k(b)$ already depends on B's Gödel number. However, B is able to compute a once it has its own Gödel number $z = b$ since step 2 of its algorithm yields $a = k(b)$.

Thus, we see that, after the execution of A on input y, B receives input $z = b$ which gives

$$a = k(z) = k(b),$$

and so B outputs into R_1 the value

$$e = \gamma(\gamma^{-1}(a), \gamma^{-1}(b), \gamma^{-1}(c)) = \gamma(ABC) = \gamma(P).$$

The following diagram shows the results of all three programs combined in sequence, where $v \xrightarrow{X} w$ means that program X inputs v and outputs w. Then we have

$$y \xrightarrow{A} (b, y) \xrightarrow{B} (e = \gamma(ABC), y) \xrightarrow{C} f(e = \gamma(ABC), y).$$

Therefore, $P = ABC = P_e$ computes

$$\phi_e(y) = f(e, y),$$

and the proof is complete. $\square$

Example 2. Program P is called **totally introspective** iff, on input y, P returns a number that encodes every configuration of the computation of itself on input y. Letting $\sigma(x, y, i)$ denote the encoding of the i th configuration of the computation $P_x(y)$, then we define the computable function

$$f(x, y) = \begin{cases} \tau(\sigma(x, y, 0), \sigma(x, y, 1), \ldots, \sigma(x, y, t)) & \text{if } P_x(y) \text{ halts in } t \text{ steps} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Now, by the 2nd recursion theorem, there exists a Gödel number e for which $\phi_e(y) = f(e, y)$, meaning that P_e is totally introspective, since, on input y, if $P_e(y)$ is defined, then P_e outputs a τ-encoding of all the configurations used in its computation $P_e(y)$! $\square$

The `self` Programming Statement

The Recursion theorem gives rise to a tool that may be used when writing a program P. Namely, we may make reference to P's Gödel number, which is represented with the keyword `self`. This allows for programs to become more *autonomous* and *self-adaptable* to its environment. For example, a program can be made to analyze its own data, make adjustments to its algorithm, followed by re-compilation and execution.

Example. The following are valid programming statements for program P.

```
void f(unsigned int y)
{
    if(y == 0) {print("bad input!\n"); return;}

    int length = instructions(self).length;
    print("Hi! I have Godel number equal to ");
    print(self);
    print(".\nI have ");
    print(length);
    print(" instructions ");

    if(y > length)
    {
        print(" which is fewer than your input ");
        print(y);
    }
    else
    {
        print("My instruction number ");
        print(y);
        print(" is ");
        print(to_string(instructions(self)[y-1]));
    }

     print("\n");
}
```

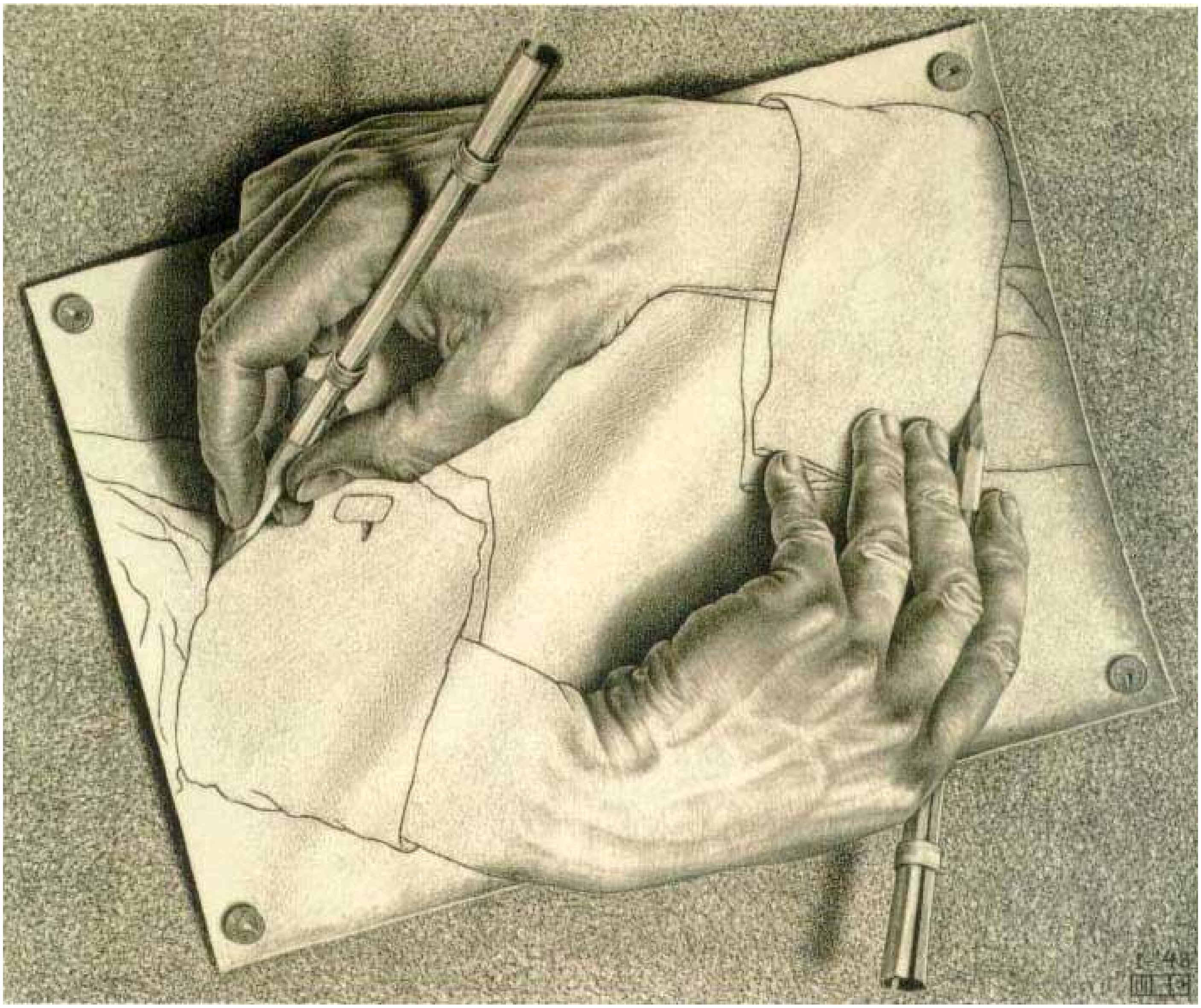
To justify such a program, suppose $y \in \mathcal{N}$ is the input to P, and the purpose of P is to implement the unary computable function $f(y)$. Then we may do the following.

1. Transform P by adding another input x, so that we are now implementing function $f(x, y)$.
2. Replace each occurrence of `self` with x.

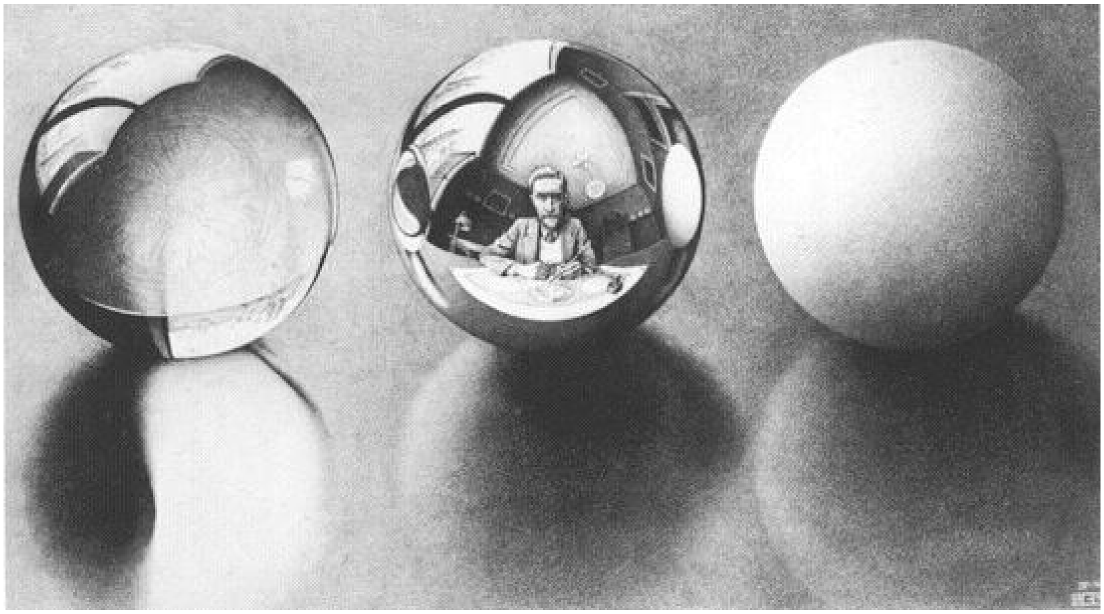
```
void f(unsigned int x, unsigned int y)
{
    if(y == 0) {print("bad input!\n"); return;}

    int length = instructions(x).length;
    print("Hi! I have Godel number equal to ");
    print(x);
    print(".\nI have ");
    print(length);
    print(" instructions ");

    if(y > length)
    {
        print(" which is fewer than your input ");
        print(y);
    }
    else
    {
        print("My instruction number ");
        print(y);
        print(" is ");
        print(to_string(instructions(x)[y-1]));
    }

     print("\n");
}
```

3. Use the method described in the proof of Kleene's 2nd Recursion Theorem to compute an e for which P_e computes
$$\phi_e(y) = f(e, y).$$
4. Thus, P_e computes $f(y)$, with e substituted for x.
5. Therefore, P_e's references to `self` are justified, since `self` $= e$, the Gödel number of the program that computes $f(y)$.

1 Self Reference Portrayed in Art and Mathematics



M.C. Escher's "Drawing Hands". 1948



M.C. Escher's "Three Spheres". 1946

Kurt Gödel: First-Order Peano Arithmetic (FOPA) is incomplete (i.e. not all true statements in FOPA can be proven true) since there is a logical statement that can be expressed within FOPA and that asserts it own unprovability within FOPA.

Kleene's 2nd Recursion Theorem and Undecidability

The `self` programming construct that is made possible by Kleene's 2nd Recursion theorem may be readily used to prove the undecidability of most program properties, including the properties `Self Accept`, `Halting Problem`, `Total`, and `Zero` from the Undecidability and the Diagonalization Method lecture. The idea is outlined as follows.

1. Let A be a program property that we want to prove is undecidable.
2. Let $d_A(x)$ denote A's decision function.
3. Assume A is decidable in which case $d_A(x)$ is total computable.
4. Consider the following program P.

   Input $y \in \mathcal{N}$.
   If $d_A(\texttt{self}) = 1$, $//P$ has property A.
       Return a value that implies P does *not* have property A.
   Else $//d_A(\texttt{self}) = 0$ and thus P does not have property A.
       Return a value that implies P *does* have property A.

5. Regardless of whether or not P has property A, a contradiction arises. Therefore, the assumption that A is decidable must be false.

Example 3. We prove that `Halting Problem` is undecidable.

Solution. Suppose `Halting Problem` is decidable, i.e.

$$H(x, y) = \begin{cases} 1 & \text{if } y \in W_x \\ 0 & \text{otherwise} \end{cases}$$

is total computable. Now consider the following program P.

> Input $y \in \mathcal{N}$.
>
> If $H(\texttt{self}, y) = 1$, loop forever.
>
> Return 1.

Let $e = \texttt{self}$ denote the Gödel number for P. Then $P_e(e) = 1$ provided $H(e, e) = 0$ iff $P_e(e)$ does not halt, a contradiction. Similarly, $P_e(e)$ does not halt provided $H(e, e) = 1$ iff $P_e(e)$ does halt, another contradiction. Therefore, the assumption that `Halting Problem` is decidable must be false.

Example 4. Prove that the `Total` decision problem is undecidable. Also, give examples of programs P_1 and P_2 for which $d_{\text{Total}}(\gamma(P_1)) = 1$ and $d_{\text{Total}}(\gamma(P_2)) = 0$.

Solution.

Example 4b. An instance of the decision problem `One-to-One` is a Gödel number x, and the problem is to decide if function ϕ_x is a one-to-one function, meaning that, for every z in the range of ϕ_x, there is *exactly one* y for which $\phi_x(y) = z$. Consider the `One-to-One` decision function

$$g(x) = \begin{cases} 1 & \text{if } \phi_x \text{ is one-to-one} \\ 0 & \text{otherwise} \end{cases}$$

Evaluate $g(x)$ for each of the following Gödel number's x.

1. $x = e_1$, where e_1 is the Gödel number of the program that computes the function $\phi_{e_1}(y) = \mathrm{sgn}(y)$. Hint: recall that $\mathrm{sgn}(y)$ equals 1 if $y > 0$, and equals 0 otherwise.
2. $x = e_2$, where e_2 is the Gödel number of the program that computes the function $\phi_{e_2}(y) = y^2$.
3. $x = e_3$, where e_3 is the Gödel number of the program that computes $g(x)$ (assuming that $g(x)$ is URM computable).

Prove that $g(x)$ is not URM computable. In other words, there is no URM program that, on input x, always halts and either outputs 1 or 0, depending on whether or not ϕ_x is a one-to-one function. Do this by writing a program P that uses g and makes use of the `self` programming construct. Then show how P creates a contradiction.

Other Applications of Kleene's 2nd Recursion Theorem

A subset $A \subset \mathcal{N}$ of the natural numbers is said to be **recursively enumerable** iff there is a program that can print all the members of A in a (possibly infinite) list, in no particular order. Also, we say that decision problem A is recursively enumerable if the set of positive instances of A is recursively enumerable.

Note: A is recursively enumerable iff there is a total computable function f for which $A = \text{range}(f)$.

Example. Show that the set of even natural numbers is recursively enumerable.

Solution. The following program prints all even natural numbers.

Input $x \in \mathcal{N}$.

For each $i = 0, 1, \ldots$

Print 2i.

Theorem. If decision problem A is decidable, then it is recursively enumerable.

Proof. Let $d_A(x)$ denote A's decision function. Since A is decidable there is a program P that halts on all inputs, and for which $P(x) = d_A(x)$ for all $x \in \mathcal{A}$. Then the following program prints all the positive instances of A.

For each $i = 0, 1, \ldots,$

> Simulate P on input i.
>
> If $P(i) = 1$, then print i.

Example. Show that `Self Accept` is recursively enumerable, i.e. we can print the set $\{i | P_i(i) \downarrow\}$.

Solution. The idea is to simultaneously simulate *all* computations $P_i(i)$, $i \geq 0$. This is accomplished by breaking up the process into rounds $0, 1, 2, \ldots$ where in Round i we perform a simulation step for each of $P_0(0), \ldots, P_i(i)$. The following program does this.

Initialize infinite Boolean array `printed` so that `printed`$[i] = 0$, for all $i = 0, 1, \ldots$.

Initialize infinite `Configuration` array `config` so that `config`$[i] = \emptyset$, for all $i = 0, 1, \ldots$.

For each $i = 0, 1, \ldots$,

    For each $j = 0, 1, \ldots, i$,

        If `printed`$[j] = 1$, then continue. //j has already been printed
        If $j < i$, then
            If `is_final_config`$[j]$, then
            1. Print j.
            2. `printed`$[j] = 1$
            Else `config`$[j] =$ `next_config`$(j,$ `config`$[j])$.
        Else `config`$[i] =$ `initial_config`(i).

Program P_x is said to **minimal** iff there is no $y < x$ for which $\phi_y = \phi_x$. In other words, x is an index for ϕ_x and there is no smaller index.

Example. Complete the following table.

Gödel Number/index	**Program**	**Function**	**Minimal?**
0	$P_0 = Z(1)$	$\phi_0(z) = 0$	Yes
1	$P_1 = S(1)$	$\phi_1(z) = \cancel{z}$ $z+1$	Yes
2	$P_2 = T(1,1)$	$\phi_2(z) = z$	Yes since P_2 is first program to compute $\phi(z) = z$
3	$P_3 = J(1,1,1)$	$\phi_3(z) = \uparrow$	Yes
4	$P_4 = Z(2)$	$\phi_4(z) = z$	No since P_2 already computes
5	$P_5 = S(2)$	$\phi_{\cancel{4}\,5}(z) = z$	No

Theorem 3. If M denotes the set of all Gödel numbers x for which P_x is minimal, then W is *not* recursively enumerable.

Proof of Theorem 3. Suppose M is recursively enumerable. Then it is an exercise to show that there is a total computable unary function f whose range is equal to M. In other words $M = \{f(i) | i \in \mathcal{N}\}$. Consider the following program P.

P :

> Input $x \in \mathcal{N}$.
>
> For each $i = 0, 1, \ldots$
>
> > If $f(i) \geq$ `self`, then `break`.
>
> Simulate program $P_{f(i)}$ on input x, and return y in case $P_{f(i)}(x) \downarrow y$.

Let e be the Gödel number of P. Then it follows that $\phi_e = \phi_{f(i)}$. But $f(i) > e$ which contradicts the fact that $f(i) \in M$. Therefore, the assumption that M is r.e. must be false.

Theorem 4. Let f be a total computable unary function. Then there is a number $n \in \mathcal{N}$ for which $\phi_n = \phi_{f(n)}$. We refer to n as a **fixed point** for f.

Proof of Theorem 4. Consider the following program P.

$f(n) = n^2$

Generates indices $0, 1, 4, 9, 16, 25, 36, 49, \ldots$

Input $x \in \mathcal{N}$.

Compute $y = f(\texttt{self})$.

Simulate program P_y on input x, and return z in case $P_y(x) \downarrow z$.

Then

$$\phi_y = \phi_{f(\texttt{self})} = \phi_{\texttt{self}},$$

and so $n = \texttt{self}$ is a fixed point for f.

An Application to Complexity Theory

$g(x, y)$

$\phi_{h(x)}(y) = g(x, y)$

The `self` programming construct may be applied to obtain a relatively simple proof of a fundamental theorem in complexity theory called the *Time Hierarchy Theorem*.

Time Hierarchy Theorem. Let $t(n) \geq n \log n$ be a computable function, for which the value $t(n)$ may be computed in $\mathrm{O}(t(n))$ steps. Then there is a decision problem L that may be decided in $\mathrm{O}(t(n))$ steps, but cannot be decided in $o(t(n)/\log n)$ steps.

$\phi_{h(n)}(x) = \longrightarrow \lfloor x^{1/n} \rfloor$

Corollary. For any positive integer $k \geq 2$, there is a decision problem that can be decided in $\mathrm{O}(n^k)$ steps, yet cannot be decided in $\mathrm{O}(n^{k-1})$ steps.

For example, there is a decision problem that can be decided within a cubic (i.e. $\mathrm{O}(n^3)$) number of steps, yet cannot be decided within a quadratic (i.e. $\mathrm{O}(n^2)$) number of steps.

Exercises

1. With respect to Kleene's 2nd Recursion Theorem, prove that there are infinitely many values e for which $\phi_e(y) = f(e, y)$. Hint: consider program B in the proof of the theorem.

2. Recall that a function $f : \mathcal{N} \to \mathcal{N}$ is **onto** provided for every $y \in \mathcal{N}$ there is an $x \in \mathcal{N}$ for which $f(x) = y$. Consider the function

$$g(x) = \begin{cases} 1 & \text{if } \phi_x \text{ is onto} \\ 0 & \text{otherwise} \end{cases}$$

   Evaluate $g(a)$, $g(b)$, and $g(c)$, where

   (a) $\phi_a(y) = y^2$

   (b) $\phi_b(y) = 1$

   (c) $\phi_c(y) = y$.

3. Prove that the function

$$g(x) = \begin{cases} 1 & \text{if } \phi_x \text{ is onto} \\ 0 & \text{otherwise} \end{cases}$$

   is not URM computable. In other words, there is no URM program that, on input x, always halts and either outputs 1 or 0 as output, depending on whether or not ϕ_x is onto. Do this by writing a program P that uses g and makes use of the `self` programming construct.

4. Recall that W_x denotes the domain of the function $\phi_x(y)$, i.e. the natural number inputs y to ϕ_x for which $\phi_x(y)$ is defined. Consider the function

$$g(x) = \begin{cases} 1 & \text{if } W_x = \emptyset \\ 0 & \text{otherwise} \end{cases}$$

   Evaluate $g(a)$, $g(b)$, and $g(c)$, where

   (a) $P_a = S(2), S(2), S(1), J(1, 2, 6), J(1, 1, 3)$

   (b) $P_b = S(2), J(2, 3, 3), J(1, 1, 1)$

   (c) $P_c = S(1), S(1), S(2), J(1, 2, 6), J(1, 1, 1)$

5. Prove that the function

$$g(x) = \begin{cases} 1 & \text{if } W_x = \emptyset \\ 0 & \text{otherwise} \end{cases}$$

   is not URM computable. In other words, there is no URM program that, on input x, always halts and either outputs 1 or 0 as output, depending on whether or not ϕ_x has an empty domain. Do this by writing a program P that uses g and makes use of the `self` programming construct. Then show how P creates a contradiction.

6. Consider the function

$$g(x) = \begin{cases} 1 & \text{if } |E_x| = \infty \\ 0 & \text{otherwise} \end{cases}$$

   In other words $g(x) = 1$ iff function $\phi_x(y)$ has an infinite range, meaning that it outputs an infinite number of different values. Evaluate $g(a)$, $g(b)$, and $g(c)$, where

(a) $\phi_a(y) = y^2$

(b) $\phi_b(y) = y$

(c) $\phi_c(y) = \text{sgn}(y)$.

7. Prove that the function

$$g(x) = \begin{cases} 1 & \text{if } |E_x| = \infty \\ 0 & \text{otherwise} \end{cases}$$

is not URM computable. In other words, there is no URM program that, on input x, always halts and either outputs 1 or 0 as output, depending on whether or not ϕ_x has an infinite range. Do this by writing a program P that uses g and makes use of the `self` programming construct. Then show how P creates a contradiction.

8. Rice's theorem states that if $\mathcal{C}_1$ denotes the set of unary computable functions, and $\mathcal{B}$ is a nonempty proper subset of $\mathcal{C}_1$, then the predicate function

$$B(x) = \begin{cases} 1 & \text{if } \phi_x \in \mathcal{B} \\ 0 & \text{otherwise} \end{cases}$$

is undecidable. Prove Rice's theorem by writing an informal program P that uses $B(x)$ and makes use of the `self` programming construct. Then show how P creates a contradiction. Hint: assume $B(x)$ is decidable, and take advantage of the fact that the set of functions $\mathcal{B}$ is both nonempty and not all of $\mathcal{C}_1$.

9. For each constant $n \geq 1$, show that $\lfloor x^{1/n} \rfloor$ is a primitive-recursive function of x.

10. Prove that there exists an n for which $\phi_n(x) = \lfloor x^{1/n} \rfloor$. Hint: use the s-m-n theorem and Theorem 4.

11. Recall that program P_x has the self-output property iff $x \in E_x$. By writing an informal program that makes use of the programming construct `self`, prove that the self-output property is undecidable.

12. Show that there is a number e for which $\phi_e(x) = e^{10}$, for all $x \in \mathcal{N}$.

13. Consider the following description of a function $f(n)$. On input n, return the Gödel number of the program P' that is the result of appending program P_n with a minimum number of successor instructions $S(1), \ldots, S(1)$ so that it is always guaranteed that, should P_n halt on an input, then the final instruction of P' will be one of these successor instructions. Then by the Church-Turing thesis, f is total computable. Moreover, prove that, if n is a fixed point for $f(n)$, i.e. $\phi_n = \phi_{f(n)}$, then necessarily $\phi_n(x)$ is undefined for all x.

Exercise Solutions

1. Since the proof of Kleene's 2nd Recursion Theorem constructs e as $e = \gamma(ABC)$, by changing the instructions of B, we get a new value for e, since B has changed. We only have to make sure that B's instructions are changed in a trivial way that does not affect its functionality as described in the proof.

2. A function $\phi_x(y)$ is onto iff $E_x = \mathcal{N}$, where E_x denotes the range of ϕ_x. Thus,

   (a) $g(a) = 0$ since $\phi_a(y) = y^2$ is not onto since $E_a = \{1, 4, 9, 25, \ldots\} \neq \mathcal{N}$,

   (b) $g(b) = 0$ since $\phi_b(y) = 1$ is not onto since $E_b = \{1\} \neq \mathcal{N}$, and

   (c) $g(c) = 1$ since $\phi_c(y) = y$ is onto since $E_c = \mathcal{N}$.

3. We have the following program P.

   Input $y \in \mathcal{N}$.

   If $g(\texttt{self}) = 1$, loop forever.

   Return y;

   If $g(\texttt{self}) = 1$, then P has a range equal to $\mathcal{N}$ which is impossible since it does not terminate on any input (loops forever). If $g(\texttt{self}) = 0$, then P does not have a range equal to $\mathcal{N}$, which is contradicted by the fact that P returns y on input y, and so has the set of return values $\{0, 1, \ldots\} = \mathcal{N}$.

4. We have the following answers.

   (a) $g(a) = 0$ since P_a terminates on input 1 (verify!) and thus $W_a = \{1\} \neq \emptyset$.

   (b) $g(b) = 1$ since P_b does not terminate on any input (why?) and thus $W_b = \emptyset$.

   (c) $g(c) = 1$ since P_c does not terminate on any input (why?) and thus $W_c = \emptyset$.

5. We have the following program P.

   Input $y \in \mathcal{N}$.

   If $g(\texttt{self}) = 1$, Return 0.

   Loop Forever.

   If $g(\texttt{self}) = 1$, then it means $W_{\text{self}} = \emptyset$, but P returns 0 for each input y, which implies $W_{\text{self}} = \mathcal{N}$, a contradiction.

   If $g(\texttt{self}) = 0$, then it means $W_{\text{self}} \neq \emptyset$, but P loops forever on each input y, which implies $W_{\text{self}} = \emptyset$, a contradiction.

6. We have the following answers.

   (a) $g(a) = 1$ since $\phi_a(y) = y^2$ has an infinite range: $E_a = \{1, 4, 9, 25, \ldots\}$,

   (b) $g(b) = 1$ since $\phi_b(y) = y$ has an infinite range $E_b = \mathcal{N}$, and

   (c) $g(c) = 0$ since $\phi_c(y) = \text{sgn}(y)$ has finite range equal to $\{0, 1\}$.

7. Consider the following program P.

   Input $y \in \mathcal{N}$.

   If $g(\texttt{self}) = 1$, Return 0.

   Return y.

If $g(\texttt{self}) = 1$, then it means $|E_{\text{self}}| = \infty$, but the program returns 0 for each input y, which implies $E_{\text{self}} = \{0\}$ which is finite, a contradiction.

If $g(\texttt{self}) = 0$, then it means $|E_{\text{self}}|$ is finite, but the program returns y on each input y, which implies $E_{\text{self}} = \mathcal{N}$, a contradiction.

8. Assume $B(x)$ is decidable. Since $\mathcal{B}$ is nonempty there exists a unary computable function $f \in \mathcal{B}$. Similarly, since $\mathcal{B}$ is not all of $\mathcal{C}_1$, there is a unary computable function $g \notin \mathcal{B}$. Now consider the following program P.

   Input $x \in \mathcal{N}$.

   If $B(\texttt{self}) = 1$,

   Simulate g on input x.

   Return $g(x)$ if it is defined.

   Simulate f on input x.

   Return $f(x)$ if it is defined.

   Since f and g are computable, so is P. Let e denote the Gödel number of P. Assume $B(e) = 1$. By definition, this means that $\phi_e \in \mathcal{B}$. But in examining P we see that P simulates g so that $\phi_e = g \notin \mathcal{B}$, a contradiction. Similarly, if $B(e) = 0$, then $\phi_e \notin \mathcal{B}$. But in this case P simulates f so that $\phi_e = f \in \mathcal{B}$, a contradiction. Therefore, B cannot be decidable.

9. The function $\lfloor x^{1/n} \rfloor$ may be computed as

$$\mu(z \le x)(z^n > x) - 1.$$

10. Function $f(n, x) = \lfloor x^{1/n} \rfloor$ is computable by the previous exercise. Therefore, by the s-m-n theorem, there exists a total computable function $k(n)$ for which $\phi_{k(n)}(x) = \lfloor x^{1/n} \rfloor$. Finally, by Theorem 4, there is an integer n for which

$$\phi_n(x) = \phi_{k(n)}(x) = \lfloor x^{1/n} \rfloor.$$

11. Assume $E(x)$ is decidable, where $E(x) = 1$ iff $x \in E_x$. Now consider the following program P.

   Input $x \in \mathcal{N}$.

   If $E(\texttt{self}) = 1$,

   Loop forever.

   Return `self`.

   Since $E(x)$ is decidable, P is computable. Let e denote the Gödel number of P. Assume $E(e) = 1$. By definition, this means that $e \in E_e$, meaning that P returns e on some input x. However, since $E(e) = 1$, P does not terminate on any input, meaning that $E_e = \emptyset$, a contradiction.

   Similarly, if $E(e) = 0$, then $e \notin E_e$. But in this case P returns e, meaning that $e \in E_e$, a contradiction. Therefore, $E(x)$, i.e. the Self-Output property, is not decidable.

12. Function $f(y, x) = y^{10}$ is primitive recursive, and hence computable. Therefore, by the s-m-n theorem, there exists a total computable function $k(y)$ for which $\phi_{k(y)}(x) = y^{10}$. Finally, by Theorem 4, there is an integer e for which

$$\phi_e(x) = \phi_{k(e)}(x) = e^{10}$$

for all $x \in \mathcal{N}$.

13. Since $f(n)$ is total computable, by Theorem 4 there is an integer n for which $\phi_n(x) = \phi_{f(n)}(x)$ for all $x \in \mathcal{N}$. But the way in which Gödel number $f(n)$ is constructed is such that, whenever $\phi_n(x) = y$ is true, then P_n halts, which in turn implies that $P_{f(n)}$ halts with $\phi_{f(n)}(x) = y + 1$, since $P_{f(n)}$ is the same as P_n, except that in its final instruction it adds 1 to register R_1. Thus, if $\phi_n(x)$ is defined, then we have $\phi_n(x) = y \neq \phi_{f(n)}(x) = y + 1$. Therefore, we must conclude that $\phi_n(x)$ must always be undefined, meaning that $W_n = \emptyset$.