Direct proofs of Rice's Theorem

Armando B. Matos

2014 and 2020

Abstract

To our knowledge most proofs of Rice's Theorem are based on a reduction of the halting problem (or other unsolvable problem) to an eventual algorithm that decides a non trivial property $\mathcal{P}$. Until recently we thought that it was the only possible kind of proof, but we found a direct diagonal proof. Later we noticed an Wikipedia entry containing still another proof (based on the Recursion Theorem). In this note we present and compare these proofs.

1 Introduction

In the literature, the proof of Rice's Theorem usually consists in proving an implication of the form:

> For any non trivial property $\mathcal{P}$ of functions:
> the existence of a decision algorithm `H` for $\mathcal{P}$ implies the existence of a decision algorithm for the halting problem.

- The hypothetical decision procedure `H` receives as input the description (or its Gödel number) of a Turing machine that implements some partial function f and outputs "yes" if the function f has the property $\mathcal{P}$ and "no" otherwise.
- $\mathcal{P}$ is a mathematical property of functions, that is, say, a first order logic formula that can include constants, variables and the function f.

In most proofs of Rice's Theorem, the halting problem, or possibly other unsolvable problem, is "reduced" to an eventual algorithm that decides some non trivial property $\mathcal{P}$. This is the kind of proof used in many references like [Odi89, Phi92, Mor98, Jon97].

Until recently we thought that it was the only possible kind of proof, but we discovered a direct diagonal proof (Section 3.1, page 4). Later we noticed in the Wikipedia another "direct" proof (Section 3.2, page 7). Still another direct proof was presented by Hamkins in an

answer to a post in `mathoverflow` (Section 3.3, page 8). All these proofs are described in this note. A "classical" proof is presented first (Section 2, page 2).

Notation. If e is the Gödel number of Turing machine, $\{e\}$ or φ_e denotes the mathematical function that it implements. HP denotes the halting problem. "iff" stands for "if and only if".

2 The "classical" proof: reduction from the halting problem

As mentioned in Section 1 (page 1), this is the kind of proof that is usually presented in the textbooks..., the proof as I knew it!

First a condensed form of the proof.

> Condensed proof: without loss of generality suppose that the completely undefined function $\uparrow(x)$ has the property $\mathcal{P}$, and let $g(x)$ be a function not satisfying the property $\mathcal{P}$. The instance is the Gödel number e of some Turing machine. Define the machine with Gödel number e' that: (i) it runs[1] $\varphi_e(e)$ and, if it halts, (ii) it runs $g(x)$. The function $\varphi_{e'}(x)$ has property $\mathcal{P}$ iff $\varphi_e(e)$ diverges. □

Now an expanded version.
Let us now assume that $Q(e)$ is an algorithm that decides if the function with index e has the property $\mathcal{P}$. That is, Q reads e, and decides (output 0 or 1) if the corresponding function has the property $\mathcal{P}$.

Let n be the index of an algorithm that never halts. Without loss of generality we may assume that $Q(n) = 0$ ($\mathcal{P}$ does not hold for the function $\varphi_n = \uparrow$).

Since $\mathcal{P}$ is a non-trivial property, it follows that there is an index m such that $Q(m) = 1$ ($\mathcal{P}$ holds).

[1] At this stage the input x is ignored

We can then define an algorithm $H(a, i)$ – which, as we will see, would decide the halting problem for $\varphi_a(i)$ - as follows:

1. Construct an index t of a Turing machine that implements the following algorithm $T(j)$:
   (a) Simulate the computation of $\varphi_a(i)$ (this may loop forever).
   (b) Simulate the computation of $\varphi_m(j)$ and returns its result.
2. Return $Q(t)$.

We can now show that H decides the halting problem "does $\varphi_a(i)$ halt?".

– Assume that the algorithm represented by a halts on input i, that is, $\varphi_a(i)$ halts. In this case $\varphi_t = \varphi_m$ (equality of functions) and, once $Q(m) = 1$ ($\mathcal{P}$ holds for m) and the output of $Q(x)$ depends only on x, it follows that:
$Q(t) = 1$ ($\mathcal{P}$ holds) and, therefore $H(a, i) = 1$ ($\varphi_a(i)$ halts).

– Assume that the algorithm represented by a does not halt on input i. In this case $\varphi_t = \varphi_n$, (it is the partial function undefined on all points). Since $Q(n) = 0$ ($\mathcal{P}$ does not hold) and the output of $Q(x)$ depends only on x, it follows that:
$Q(t) = 0$ ($\mathcal{P}$ does not hold) and, therefore $H(a, i) = 0$ ($\varphi_a(i)$ does not halt).

Since the halting problem is known to be undecidable, this is a contradiction – it is decided by $H(a, i)$ - and the assumption that there is an algorithm $Q(x)$ that decides a non-trivial property ($\mathcal{P}$) of the function represented by x must be false.

3 Proofs without assuming the undecidability of the halting problem

3.1 A proof based on a Turing machine construction

3.1.1 Non trivial properties

Let $\mathcal{P}$ be a non-trivial property of partial recursive functions. Let `Y(x)` and `N(x)` be two programs[2] (which are strings), such that the function implemented by `Y(x)` has the property $\mathcal{P}$, and the function implemented by `N(x)` does not have the property $\mathcal{P}$.

3.1.2 Diagonal proof of Rice Theorem

Assume that it is decidable whether a program with a given text defines a function that has the property $\mathcal{P}$. This means that there is a total program $\texttt{H} : \Sigma^* \to \Sigma^*$ such that[3]

$$\texttt{H(x)} = \begin{cases} \text{“}\texttt{yes}\text{”} & \text{if the program } \texttt{x} \text{ has property } \mathcal{P} \\ \text{“}\texttt{no} & \text{if the program } \texttt{x} \text{ does not have property } \mathcal{P} \end{cases} \quad (1)$$

Thus, `H(N)` =“`no`” and `H(Y)` =“`yes`”.

Define the program $A(x)$ with input x by

$$A(x) = \begin{cases} \texttt{if } H(\text{A}) = \text{“}\texttt{yes}\text{”} \texttt{ then } N(x) \\ \texttt{else } Y(x) \end{cases} \quad (2)$$

where “A” is the program that implements A itself. Note that in an universal Turing machine $A(x)$ can find “A” from the contents of the tape which has the form “$\ldots$A$\ldots x \ldots$”. The programs H, N and Y must of course be included explicitly in the program of A. If A has the property $\mathcal{P}$, $H(\text{A}) =$ “`yes`”, and $A(x) \equiv N(x)$, and, b y assumption, $N(x)$ does not have the property $\mathcal{P}$. And we find a similar situation if A does not have the property $\mathcal{P}$. Thus there is a

[2] Instead of programs and strings we can of course use Turing machines and Gödel numbers.

[3] Strings that do not represent a program (illegal syntax) are irrelevant. The string outputted by the function $H(x)$ is of no importance.

contradiction and the program (or Turing machine) H can't exist. □

3.1.3 The proof in more detail...

We will use some fixed universal Turing machine (UTM) as the standard "mechanical computing device". Let us agree on the following format for the UTM:

- The input alphabet is $\{a, b\}$ and the tape alphabet is $\{\text{▫}, a, b\}$ where "▫" denotes the blank.
- Initially, the tape contents is as follows "$\ldots$▫▫P▫x▫▫$\ldots$" where P is the program being executed and x its input. Both P and x are words with alphabet $\{a, b\}^*$. The machine head is at the leftmost symbol of x.
- When and if the computation finishes, the tape contents has the form "$\ldots$▫▫P▫x▫y▫▫$\ldots$", where the machine head is at the leftmost symbol of y.

Assume that there is a program `H` that determines whether the program x has the property $\mathcal{P}$. The initial configuration is "$\ldots$▫▫`H`▫x▫▫$\ldots$" and the computation halts with output "`a`" or "`b`" meaning that the program x has or does not have the property $\mathcal{P}$ respectively:

$$\ldots\text{▫▫}\texttt{H}\text{▫}x\text{▫}\texttt{a}\text{▫▫}\ldots \quad \text{or} \quad \ldots\text{▫▫}\texttt{H}\text{▫}x\text{▫}\texttt{b}\text{▫▫}\ldots$$

In order to implement the function $A(x)$ (equation (2), page 4) consider a slight modification of `H`, denoted by `H'`, that does not "look" at the string x, but instead analyses the string `H'` itself. More precisely, the program (string) which is analysed is obtained as follows, assuming that the head is located within the program to be analysed.

- Let h be the location of the UTM head.

Time	Tape contents	This example
Initial:	$\ldots\square\square\ \overbrace{\underline{\mathtt{g}}\mathtt{H'YN}}^{\mathtt{P}}\ \square\,\mathtt{x}\,\square\,\square\ldots$	
After $\mathtt{H'(P)}$:	$\ldots\square\square\ \overbrace{\mathtt{gH'YN}}^{\mathtt{P}}\ \square\,\mathtt{x}\,\square\,\underline{\mathtt{a}}\,\square\,\square\ldots$	$\mathtt{H'(P)}$ = "a"
Final:	$\ldots\square\square\ \overbrace{\mathtt{gH'YN}}^{\mathtt{P}}\ \square\,\mathtt{x}\,\square\ \overbrace{\underline{\mathtt{b}}\mathtt{aa}}^{\mathtt{y}}\ \square\,\square\ldots$	$\mathtt{N(x)}$ = "baa"

Figure 1: Three snapshots of a Turing machine (TM) computation. This TM implements a function equal to $N(x)$ if the function that it implements ($A(x)$) has the property $\mathcal{P}$, and implements a function equal to $Y(x)$ if the function that it implements does not have the property.

- Go to the left until the first blank is found. Let this location be l.
- Go (from l) to the right until the first blank is found. Let this location be r.
- The string x (representing a Turing machine) which will be analysed by $\mathtt{H'}$ is described by the (possibly empty) segment of tape from $l+1$ to $h-1$.

From H', the programs Y and N mentioned above, and of course the UTM, we define a new Turing machine that implements $A(x)$, (2), page 4. Three snapshots of a possible execution of this Turing machine for program 2 are sketched in Figure 3.1.3 (page 6), where, by convention, $\mathtt{H'(P)}$ ="a" means "yes" (P has property $\mathcal{P}$) and $\mathtt{H'(P)}$ ="b" means "no". In the example the first case applies.

The symbol scanned by the head is underlined. The part g of the program represents the coding "glue" needed to split the programs H', Y and N and to implement the program (2) (page 4), that is, $A(x) \equiv \mathtt{if}\ H'(A)\ \mathtt{then}\ N(x)\ \mathtt{else}\ Y(x)$. Note that in this case $\mathtt{H'}$ finds if the complete program A has the property $\mathcal{P}$.

In the example shown $\mathtt{H'(P)}$ ="a" ("yes"), so that $\mathtt{N(x)}$ is executed, $\mathtt{P(x)} = \mathtt{N(x)}$ ="baa"; the corresponding output is (in this example!)

"baa".

In this example the mathematical functions $\mathtt{P}(x)$ and $\mathtt{N}(x)$ are equal. Yet, $\mathtt{P}(x)$ has the property $\mathcal{P}$ (because $\mathtt{H}'(\mathtt{P}) =$"a") and by assumption $\mathtt{N}(x)$ does not have the property $\mathcal{P}$, a contradiction.

If $\mathtt{H}'(x) =$"b" ($\mathtt{P}(\mathtt{x})$ does not have the property $\mathcal{P}$) we get a similar contradiction.

The only way out of this contradiction is to suppose that, *no matter what the non trivial function property $\mathcal{P}$ is, there is no decision procedure for $\mathcal{P}$*, that is, there is no recursive function $\mathtt{H}(\mathtt{P})$ that determines if the program P implements a function with property $\mathcal{P}$.

This proof was obtained by diagonalization, without assuming the undecidability of the HP or of any other decision problem. Furthermore, and unlike the proof described in Section 3.2, we do not use "high level" recursion concepts – like Kleene's recursion theorem or quines.

3.2 A proof from the Wikipedia

In: http://en.wikipedia.org/wiki/Rice%27s_theorem

Similarly to the proof presented in Section 3.1 (page 4), the proof below, transcribed from the Wikipedia, is direct, i.e. it is not based on a reduction from an undecidable decision problem. The main argument of the proof is in the second paragraph of the quotation below.

Proof by Kleene's Recursion Theorem

Transcription:

> A corollary to Kleene's recursion theorem states that for every Gödel numbering $\varphi : \mathbb{N} \to \mathbf{P}^{(1)}$ of the computable functions and every computable function $Q(x, y)$, there is an index e such that $\varphi_e(y)$ returns $Q(e, y)$. (In the

following, we will say that f(x) "returns" $g(x)$ if either $f(x) = g(x)$, or both $f(x)$ and $g(x)$ are undefined.) Intuitively, φ_e is a quine, a function that returns its own source code (Gödel number), except that rather than returning it directly, φ_e passes its Gödel number to Q and returns the result.

Let $\mathcal{F}$ be a set of computable functions such that $\varnothing \neq \mathcal{F} \neq \mathbf{P}^{(1)}$. Then there are computable functions $f \in \mathcal{F}$ and $g \notin \mathcal{F}$. Suppose that the set of indices x such that $\varphi_x \in \mathcal{F}$ is decidable; then, there exists a function $Q(x, y)$ that returns $g(y)$ if $\varphi_x \in \mathcal{F}$, and $f(y)$ otherwise. By the corollary to the recursion theorem, there is an index e such that $\varphi_e(y)$ returns $Q(e, y)$. But then, if $\varphi_e \in \mathcal{F}$, then φ_e is the same function as g, and therefore $\varphi_e \notin \mathcal{F}$; and if $\varphi_e \notin \mathcal{F}$, then φ_e is f, and therefore $\varphi_e \in \mathcal{F}$. In both cases, we have a contradiction.

3.3 Hamkins proof

The following proof is by Joel David Hamkins and appeared an answer to a `mathoverflow` post.

Here is a proof based on the recursion theorem, rather than a reduction of an undecidable problem.

Rice's Theorem. Let $\mathcal{P}$ be a non trivial set of computable functions[4]. Then the set $\{e \mid \varphi_e \in P\}$ is not decidable, where φ_e is the function computed[5] by program e.

In other words, there is no general procedure to determine from a program whether the function it computes has property $\mathcal{P}$ or not.

[4] That is, $\mathcal{P}$ is neither empty nor the set of all computable functions.
[5] Another common notation for φ_e is $\{e\}$.

Proof. Suppose that the set were decidable. Fix a computable function f that is in $\mathcal{P}$, and another computable function g that is not in $\mathcal{P}$. Now, for any program e, let $h(e)$ be the program that on input n first determines whether $\varphi_e \in P$; if so, it outputs $g(n)$, and otherwise $f(n)$. So $\varphi_{h(e)}$ is either g or f, depending on whether $\varphi_e \in P$ or not, respectively (note that the "opposite function" is used). In particular, we'll have $\varphi_e \in P \Leftrightarrow \varphi_{h(e)} \notin P$.
Meanwhile, by the recursion theorem, there is a program e such that $\varphi_e = \varphi_{h(e)}$, which now gives an immediate contradiction, since φ_e and $\varphi_{h(e)}$ are supposed to be opposite with respect to $\mathcal{P}$. □

4 Conclusion

Rice Theorem is very useful for proving the undecidability of some property $\mathcal{P}$. One only has to show that there is at least one function that satisfies $\mathcal{P}$ and another[6]. Thus, and as there are direct proofs of Rice Theorem, it seems that a study of Recursion Theory can "begin" with such a proof, perhaps Hamkins simple proof in page 9, and later derive its consequences, namely the undecidabilty of particular problems.

References

[Jon97] Neil D. Jones. *Computability and Complexity - from a Programming Perspective*. Foundations of Computing Series. MIT Press, 1997.

[Mor98] Bernard Moret. *The Theory of Computation*. Addison-Wesley, 1998.

[Odi89] Piergiorgio Odifreddi. *Classical Recursion Theory – The Theory of Functions and Sets of Natural Numbers*, volume I. Studies in Logic and the Foundations of Mathematics. Elsevier North Holland, 1989.

[6] It should be noticed that, as often happens in Recursion Theory, none of the proofs presented in this note is constructive, essentially because we need to have two particular functions: one with the property and another without.

[Phi92] Iain Phillips. Recursion theory. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, pages 79–187. Oxford University Press, 1992.