An informal proof of Rice and Kleene theorems in Python

Massimo Santini

May 20, 2013

Abstract

In this note we give an informal proof of Rice and Kleene theorems based on the use of Python programming language instead of on formal systems like Turing machines, or partial recursive functions.

Preliminary notions

Here we restrict our attention to Python *functions* that work on *string* in the sense that their parameters and return values (in case of *termination*[1]) are of `str` type; to simplify the notation, in the following we use uppercase letters $F, G, \ldots$ to denote the functions and lowercase letters $f, g, \ldots$ to denote their *source code*. For instance, if f is:

```
def F( x ):
        return 2 * x
```

we denote with F the function that, given a string as a parameter, returns the string obtained concatenating the value of the parameter with itself; for example $F(\text{'hello'}) = \text{'hellohello'}$.

We say that two functions F and G have the *same behavior*, in symbols

$$F(x) \equiv G(x)$$

if and only if for every value of x such that $F(x)$ terminates, $G(x)$ also terminates and $F(x) = G(x)$; abusing the notation, we use it also for source code, that is, by

$$f \equiv g$$

we mean $F(x) \equiv G(x)$.

The main tools required by the first proof are: the *universal* function U and the *currying* function S; such functions can be implemented in a straightforward way in Python as follows.

[1]A function is said to terminate, on a given input, if its execution requires a finite number of steps; on the other hand, it does not terminate if, for instance, it enters an infinite loop.

The U function

The universal function, given two strings f and x as arguments, returns the string $F(x)$, where F is the function corresponding to the source code f, more formally

$$U(f, x) \equiv F(x)$$

A possible implementation of such funciton is:

```
def U( f, x ):
  locals = {}
  exec( f, globals(), locals )
  F = next( iter( locals.values() ) )
  return F( x )
```

the only notable part is the use of `exec` function that is able to execute the code represented by the string f, the rest of the implementation deals with the detail of retrieving the function from the `locals` dictionary and to compute its value.

The S function

The curring function, given the strings f and y as arguments, where f is the source code of a function accepting two parameters, returns a string g corresponding to the source code of a function G such that $F(x, y) \equiv G(x)$; more formally

$$S(f, y) = g \qquad \text{such that} \qquad U(g, x) = U(S(f, y), x) \equiv F(x, y)$$

Implementing such function is even more trivial:

```
def S( f, y ):
  n = match( 'def\s+([^(]+)\s*\(', f ).group( 1 )
  f = f.replace( '\n', '\n\t' )
  g = 'def G( x ):\n\t{0}\n\treturn {1}( x, {2!r} )'
  return g.format( f, n, y )
```

this "wraps" the function f (once getting his name and re-indenting it) obtaining a new function G of which it returns the source code.

To better understand how it works, let's apply it to the example f defined as

```
def Sum( x, y ):
        return x + y
```

that we can transform in $S(f, 3)$ as

```
def G( x ):
  def Sum( x, y ):
    return x + y
  return Sum( x, 3 )
```

Kleene’s theorem

We have all the tools required to proof Kleene’s theorem that can be stated as follows.

Teorema 1 *Given any function T that terminates for every value of its parameter, it is possible to build a function r such that $T(r) \equiv r$.*

We show a constructive proof of the theorem, that is, we build the function r using T and the two building blocks U and S introduced in the previous section. Let’s consider the functions e and m respectively defined by

```
def E( x, f ): return U( U( f, f ), x )
```

```
def M( x ): return T( S( e, x ) )
```

We now show that taking $r = S(e, m)$ gives the function named by the above theorem (observe that the source code of r contains calls to T, E, M, S and U):

```
def R( x ):
  def E( x, f ): return U( U( f, f ), x )
  return E( x, 'def M( x ): return T( S( e, x ) )' )
```

To show it simply requires to perform a series of very simple steps:

$$\begin{aligned}
U(r, x) &= U(S(e, m), x) && \text{by definition of } r,\\
&= E(x, m) && \text{by definition of } S,\\
&= U(U(m, m), x) && \text{by definition of } E,\\
&= U(M(m), x) && \text{by definition of } U,\\
&= U(T(S(e, m)), x) && \text{by definition of } M,\\
&= U(T(r), x) && \text{by definition of } r.
\end{aligned}$$

which, by definition of $\equiv$ ed U, leads to the theorem statement.

A fun application: quine

We can use the Kleene theorem to obtain a *quine*, that is, a function that returns its source code; by this we mean a function F such that $F(x)$ is equal (whatever the value of parameter x) to its source code f.

This is not a trivial task

```
def F( x ):
  return 'def F( x )\n\t return x'
```

for example, is such that $F(x)$ has value

```
def F( x ):
  return x
```

that looks like its source, but is not strictly identical to it. On the other hand, it is quite trivial to design a function T (always terminating), that given as argument a source code f returns a function g such that $G(x)$ always returns f (whatever the value of x is):

```
def T( f ):
  return 'def G( x ):\n\treturn {0!r}'.format( f )
```

Thanks to Kleene's theorem, it is now easy to obtain r such that r and $T(r)$ have the same output, that is $R(x)$ is always equal to r.

Rice's theorem

We finally get to the main goal of this note.

We say that a class $\mathcal{F}$ of functions is said to *preserves properties* if and only if

$$f \in \mathcal{F} \quad \text{and} \quad f \equiv g \qquad \text{implies} \qquad g \in \mathcal{F}$$

to put it in another way, $\mathcal{F}$ contains all the function that "have the same behavior"; we also say that such a collection $\mathcal{F}$ can be *decided* if one can write a function D that always terminates such that $D(f)$ yield `yes` or `no` whether $f \in \mathcal{F}$ or not; we call *trivial* the empty class and the class containing all the function.

We are now ready to state the theorem.

Teorema 2 *If $\mathcal{F}$ preserves properties and is not trivial, then it can't be decided*

A proof by contradiction can be obtained using the Rice theorem. Let $p \in \mathcal{F}$ e $q \notin \mathcal{F}$ (they must exists since $\mathcal{F}$ is not trivial) and let D the function that decides $\mathcal{F}$; let's consider the function T (which always terminates, given that D does):

```
def T( f ):
  if D( f ) == 'yes':
    return q
  else:
    return p
```

By the Kleene's theorem, there exists an r such that $r \equiv T(r)$ and hence, given that $\mathcal{F}$ preserves properties, it can be $r \in \mathcal{F}$ and $T(r) \in \mathcal{F}$, or $r \notin \mathcal{F}$ and $T(r) \notin \mathcal{F}$. This can't be, given that by definition of T if $r \in \mathcal{F}$ then $D(r) =$`yes` and hence $T(r) = q \notin \mathcal{F}$; on the other hand, if $r \notin \mathcal{F}$, then $D(r) =$`no` and hence $T(r) = p \in \mathcal{F}$.