Why do Quines exist: The Recursion Theorem

Dan Hathaway

What is a Quine?

- A Quine is a program that outputs its own source code
- Not immediately obvious that they exist, especially in every computer language, but they do!
- Quine in Haskell (wikipedia):

```
main=putStr(p++show(p))where p=
"main=putStr(p++show(p))where p="
```

The Recursion Theorem

- We want to prove that Quines (and things like Quines) exist and have a way to construct them
- Before we talk about proving such a result, we need a model of computations (Turing Machines, Lambda Calculus, etc).

Model Of Computation

Let $\varphi_n^{(m)}$ denote the program that takes m string arguments with source code being the string n, and let $\varphi_n^{(m)}(x_1, \ldots, x_m)$ denote the "output"* of the program when passed the strings $x_1, \ldots, x_m$.

*Note: In this model of computation, a program must halt in order for the output to count. If a program $\varphi_n^{(m)}$ does not halt on input $x_1, \ldots, x_m$ we write $\varphi_n^{(m)}(x_1, \ldots, x_m) \uparrow$.

$\forall_{x_1,\ldots,x_m} \varphi_{n_1}^{(m)}(x_1, \ldots, x_m) = \varphi_{n_2}^{(m)}(x_1, \ldots, x_m)$ means that for each input $x_1, \ldots, x_m$ either both $\varphi_{n_1}^{(m)}$ and $\varphi_{n_2}^{(m)}$ do not halt or they both halt and output the same string.

Important Lemma

(s-m-n Theorem):

For all e, m, n there exists an s such that for all $y_1, ..., y_m$ we have $\varphi_s^{(m)}(y_1, ..., y_m) \downarrow$ and for all $x_1, ..., x_n$

$$\varphi_e^{(m+n)}(y_1, ..., y_m, x_1, ..., x_n) = \varphi^{(n)}_{\varphi_s^{(m)}(y_1,...,y_m)}(x_1, ..., x_n))$$

(We will take this theorem as given for any real programming language)

Recursion Theorem (Statement)

Theorem (Recursion):

If f is any program that takes one input string and always halts, then there is some string q such that

$$\forall_{x_1,\ldots,x_m} \varphi_q^{(m)}(x_1,\ldots,x_m) = \varphi_{f(q)}^{(m)}(x_1,\ldots,x_m).$$

Example: If f is any program that takes input y and returns the source code for a program that prints y, then the q will be the source code of a Quine.

Note: This theorem is amazing. Essentially, any program that does a task can be modified into a program that does the same task but that knows about its new self (source code). However, we cannot construct programs that know in general what they will **do**, because then we could make a program that does the opposite of what it knows it will do (which is impossible).

Recursion Theorem Proof (Clever)

First, we can write a program h such that

$$\forall_{e,x_1,\ldots,x_m} h(e, x_1, \ldots, x_m) = \varphi^{(m)}_{f(\varphi^{(1)}_e(e))}(x_1, \ldots, x_m).$$

Writing h may be as difficult as writing an interpreter for our computer language in the language. Next, applying the s-m-n theorem to reduce the number of inputs to h we have that there is some s such that

$$\forall_{e,x_1,\ldots,x_m} \varphi^{(m)}_{\varphi^{(1)}_s(e)}(x_1, \ldots, x_m) = h(e, x_1, \ldots, x_m).$$

Combining the two equations and setting $e = s$:

$$\forall_{x_1,\ldots,x_m} \varphi^{(m)}_{\varphi^{(1)}_s(s)}(x_1, \ldots, x_m) = \varphi^{(m)}_{f(\varphi^{(1)}_s(s))}(x_1, \ldots, x_m).$$

Calling $q = \varphi^{(1)}_s(s)$, we have proved the theorem.

Easy to Understand Corollary:

If g is any program that takes the inputs $e, x_1, ..., x_m$ then there is some q such that

$$\varphi_q^{(m)}(x_1, ..., x_m) = g(q, x_1, ..., x_m).$$

Proof: Just apply s-m-n theorem to g to write in the form $\varphi^{(m)}_{\varphi_s^{(1)}}$ can let $f = \varphi_s^{(1)}$ and apply the recursion theorem.

All computer scientists should know this version of the theorem!

Advanced

Suppose f is any function from the set of all strings into itself and suppose that this function can be computed by a Turing machine that uses the halting set as an oracle but the halting set cannot be computed by a Turing machine that uses f as an oracle. In this case, we still have

$$\exists_q \forall_{x_1,\ldots,x_m} \varphi_q^{(m)}(x_1,\ldots,x_m) = \varphi_{f(q)}^{(m)}(x_1,\ldots,x_m).$$

That is, we can apply the recursion theorm to any function f that is NOT Turing-complete, even if f is not computable.

Note: By "not Turing-complete", I mean not every recursively enumerable set is Turing reducible to f.

Bibliography

- “Recursively Enumerable Sets and Degrees” by Robert I. Soare
- “Computable Structures and the Hyperarithmetical Hierarchy” by C.J. Ash, J.F. Knight
- Wikipedia!