Perl and Undecidability: Rice's Theorem

by Jeffrey Kegler
jeffreykegler@mac.com

This winter I moved into a cabin at the edge of a frozen lake and forsook gainful employment in favor of work on a CPAN module. Currently, no generally accepted, handy tool accepts arbitrary BNF and parses with it. Recent research suggests how to create that tool. I call my effort `Parse::Marpa`. An alpha version is now on CPAN.

Approaching beta, I thought about potential applications. My thoughts turned to Perl 5. The Perl community had reached the consensus that Perl 5 is not statically parseable. Adam Kennedy, in the documentation for PPI, indicated how this might be proved. I worked the proof out formally and posted it on Perlmonks and also published it in this journal (XXX).

The formal proof shows that Perl 5 is not just statically unparseable, it is also dynamically unparseable. The saying had been that "Only perl can parse Perl". In fact, not even Perl 5 can parse Perl 5 in every case.

Here's the reason: The only way to parse Perl 5 is to run it or to simulate it using a language of equivalent power. Perl 5 is what's called Turing-complete, and all Turing-complete languages are subject to the Halting Problem. There is no guarantee a Perl 5 program will ever finish running and no guarantee it will ever finish parsing itself.

In this article I will use Rice's Theorem to prove that Perl is unparseable. Rice's Theorem is powerful and easy to apply. It's well known in mathematical circles and deserves to be better known by programmers.

■ Undecidable parsing is a feature------------------------

Undecidable parsing is not a bug. It is not a misfeature. It goes hand-in-hand with important capabilities. Understanding this is important to understanding where programming languages are going.

Perl 5 is unparseable because it gives the programmer Turing-complete power before compile time. As time goes on, it becomes clearer and clearer that Larry Wall aimed Perl in the right direction. Theoretical perfection at compile time is a loss at run-time. Industrial strength debugging and optimization require information unavailable before run-time. Decidability is not good if the decisions are bad.

■ Decidability--

Decidability means the ability for a Turing-complete machine (or language), to determine the answer to a yes/no question. A yes/no question is decidable if a Turing-complete Perl script can answer it. Otherwise it is undecidable.

Turing-equivalence means equivalence to the model of computing in Alan Turing's 1936 paper. A machine or language is Turing-complete if it has Turing-equivalent power or better. All modern general-purpose machines and languages are at least Turing-complete.

Perl scripts differ from theoretical Turing-complete programs in two ways, neither of them serious obstacles to Rice's Theorem. First, Turing-complete machines and programs have unlimited memory. The equivalent Perl implementation would never run out of memory. A real-life Perl script will fail to answer an undecidable question by running out of memory. Its theoretical Turing-complete counterpart fails by running forever. This is not a difference we need to care about.

Second, Perl scripts have certain capabilities which Turing-equivalent machines do not. Turing-equivalent machines and languages must be completely predictable (deterministic). Perl has unpredictable features like its `rand` built-in. Pedantically, `rand` is pseudo-random instead of random, but from the point of view of the Perl script `rand` is close enough to random. Perl's ability to interact with outside processes and across networks means many Perl calls behave unpredictably, perhaps even in the quantum mechanical sense.

But unpredictability is no obstacle to undecidability proofs. An undecidable question does not become decidable when its subject matter becomes unpredictable.

■ Undecidability--

Rice's Theorem states that every interesting question about what a Perl script does is undecidable.

- Does a Perl script ever print the character '0'?
- Does a Perl script write to STDERR?
- Is a Perl script's output the same as its input?
- Does a Perl script fork a shell?
- Does a Perl script contain a virus?

All of these questions can be proved to be undecidable using Rice's Theorem. Here is Rice's Theorem more formally:

Any question about what an arbitrary Perl script does with an arbitrary input is undecidable, unless it is trivial.

■ Trivial? ---

For the purposes of Rice's Theorem, a question is trivial if the answer is always "yes", or if the answer is always "no". A trivial question is one which is true or false regardless of the Perl script I'm asking about.

An example of a trivial question is "Will the output of this Perl script be zero or more characters in length?" This answer has to be "yes". The opposite question, "Will the output of this Perl script be negative in length?", will always be "no". It also is a trivial question. The opposite of any trivial question will always be another trivial question.

■ Is versus does

For a question to be proved undecidable by Rice's Theorem, it must be about what a Perl script does. It cannot merely be about what a Perl script is. For example, Rice's Theorem applies if I am asking "Does a Perl script ever print the character '0'?". Rice's Theorem does not apply if the question is "Does a Perl script contain the character '0'?".

The mathematician's way to say this is that Rice's Theorem applies only to questions about partial functions. Ignore the "partial" in "partial function" for the moment. Programmers know what a partial function is, if not always by that name. I want to look carefully at the term "function" first.

A function is a mapping of each member of a set of inputs to exactly one output.

The requirement that each input have exactly one output is important. This Perl subroutine implements a function:

```
sub successor { (shift)+1 }
```

The answer for 1 can only be 2. The answer for -1 can only be 0. I am ignoring overflow issues. Every input to `successor` has a most one output.

Constant functions are functions which have the same output for all inputs. Here's one that always returns the same number, 42:

```
sub the_answer { 42 }
```

A function must have exactly one output for each input. The outputs do not have to be unique to each input. Constant functions, where the output is always the same, are quite acceptable as functions for the purposes of Rice's Theorem.

Perl subroutines can be functions, and are sometimes called functions. Unless I make it clear otherwise, from here on out "function" will mean the partial function performed by a Perl script.

■ Partial functions

Computer scripts, programs and subroutines are *partial* functions. A partial function is a function which might fail to produce an output.

The classic way for a program to fail to produce an output is for it to loop forever. It's a convenient example, because it's implementation-independent. Every general-purpose method of programming a computer is capable of infinite loops.

I can also say that if a Perl script returns an exit code other than zero, it fails to produce an output. If I do that, it becomes hard to take into account output from the Perl script prior to exit. How to best define output depends on the proof.

■ Predictability

One Perl script which does not implement a function is `rand`:

```
say rand(42);
```

This script fails to implement a function because, for any given input, it can produce many different outputs.

This means a Perl script with `rand` or any other unpredictable `system` call may fail to implement a function, and therefore strictly speaking will be outside the scope of Rice's Theorem. In fact, as I explained earlier, undecidability results apply as much to unpredictable scripts as to predictable ones.

■ Applying Rice's Theorem

Here's a list of five requirements, or conditions. One of the conditions is for a yes/no question. If all five conditions hold, then Rice's Theorem tells us that the yes/no question is undecidable.

1. A definition of the input to Perl scripts.
2. A definition of the output from Perl scripts.
3. A yes/no question about the output of Perl scripts. The question must look at the scripts as if they were partial functions.
4. A case of a Perl script and an input for which the answer to the question is "yes".
5. A case of a Perl script and an input for which the answer to the question is "no".

Carefully defining input and output is necessary. In some proofs, the input and output will be the only conditions that are not obvious.

■ Undecidable: Does a Perl script print 0?

1. Definition of the Input: The characters available on STDIN.

2. Definition of the Output: The characters written to STDOUT.

3. The Question to be Decided: For any input and any perl script, does it print the character '0' as part of its output?

4. A Case where the Answer is "No": The empty Perl script with empty input.

5. A Case where the Answer is "Yes": The Perl script script "`say 0`" with empty input.

The question to be decided is clearly about the output as a function. The two cases show that it is non-trivial. By Rice's Theorem the question is undecidable. QED. ("QED" is the traditional way to indicate the end of a proof.)

The "empty Perl script" is the zero length Perl script. I'll use the empty Perl script as much as I can.

The empty input is the zero length input. In this case, the question does not compare input and output.

■ Undecidable: output same as the input?

Input: The characters available on STDIN.

Output: The characters written to STDOUT.

Question: Given a Perl script, will the output ever be the same as the input?

"No" Case: The empty Perl script with any input of length greater than 0.

"Yes" Case: The empty Perl script with the empty input.

QED.

Proofs like this and the previous proof can be constructed for any non-trivial question about STDOUT as a function of STDIN.

■ Undecidable: does it write to STDERR?

Input: The characters available on STDIN.

Output: The characters written to STDERR.

Question: For any Perl script and any input, does the script with that input write to STDERR?

"No" Case: The empty Perl script.

"Yes" Case: "`say STDERR 42`".

This question is clearly about the output as a function. Rice's Theorem applies. QED.

■ Undecidable: does it fork a shell command?

Not all Perl variants are capable of `fork`'ing, and some don't have shells available. This proof requires an additional assumption:

There are Perl scripts which `fork` and `exec` shell commands, and at least one shell command can be identified by its name.

With this assumption I can proceed as usual:

Input: Characters available on STDIN (but actually not relevant).

Output: A trace of `fork` and `exec` commands, including the name of the command `exec`'d.

Question: For any Perl script and any input, does the script with that input `fork` and `exec` a shell command?

"No" Case: The empty Perl script.

"Yes" Case: By the assumption above, there is a script that `fork`'s and `exec`'s a shell command.

The question is clearly about the output, and is non-trivial. QED.

This proof uses a really nice technique which I learned from the next proof. I don't define the term "shell command". It's not necessary or useful to do so. All the proof needs is the assumption that there is such a thing as a shell command. With that I can ignore the issue of exactly what is or is not a shell.

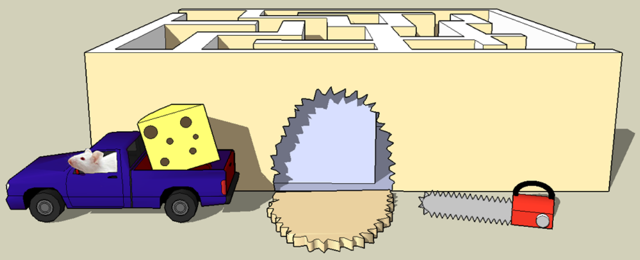
I can substitute the term "interesting command" for "shell command" in the proof and its assumption. The new proof shows that it is undecidable whether a Perl script `fork`'s and `exec`'s any interesting commands. Which commands I call "interesting" is up to me.

■ Undecidable: Does it contain a virus?

The ideas in this proof are from William Dowling. Dowling didn't define virus. He assumed a few things about viruses:

1. Viruses infect systems, in the process changing memory, disk, or some other kind of readable storage.

Sometimes you just need a little help to get ahead.



Perl trainings by author Peter Scott
All levels
On-site in North America

Pacific Systems Design Technologies
http://www.psdt.com/
250.474.0976
info@psdt.com

2. It is possible to write a Perl script that is a virus.

2. At least one Perl script is not a virus.

Input: A dump of readable storage as it existed when the Perl script started.

Output: A listing of all the changes to readable storage caused by the Perl script.

Question: For any Perl script and any input, is the Perl script with that input a virus?

"No" Case: Above, I assumed that one Perl script is not a virus. For this Perl script and some input, the answer is "no".

"Yes" Case: I assumed that a virus could be written in Perl. So for a least one Perl script and any input, the answer is "yes".

I assumed that viruses change readable storage, so the question is about the output of a Perl script as a function. All the conditions of Rice's Theorem are fulfilled. I cannot decide, in general, whether a Perl script will infect readable storage. Therefore I can't know if it's a virus. QED.

This proof looks at input and output from a different angle—as snapshots of the system. I don't deal with standard input. I've never heard of a virus that waited for a say-so from the user. If I want to allow for the possibility of polite viruses, I can modify the definition of input to include user input.

In real-life, some of the changes to readable storage won't be made by the Perl script. Even if I forbid other applications, system processes might make changes. That's why the output includes only changes to readable storage made by the Perl script. I could use tracing to determine which changes those are.

But the proof still works, even if I can never figure out which process caused what change. Here's why: From the full listing of all changes to readable storage by all processes, I can generate a set of listings, one for every possible choice among the changes. One listing in this set is the one that contains all the changes caused by the Perl script, and only those changes. The proof works if I use that one. Since I know that listing exists, I know the proof works, even if I don't know which listing the proof needs.

In all these proofs, I don't have to show how I would compute the inputs and outputs for a real-life example. The inputs and outputs have to actually exist. But I don't have to know how to compute them.

■ Does a Perl script contain a bug?

A beauty of the Virus Proof is its minimal assumptions. Anything I might want to call a virus fits the assumptions. So do a lot of things I would not call viruses. That's OK. It's just fine if the Virus Proof proves more than it sets out to prove.

Replacing the word "virus" with "bug" turns the Virus Proof into a proof that no Perl script can find all the bugs in another, arbitrary, Perl script. Once again, only a few basic assumptions about bugs are needed, not a definition.

One additional change to the Virus Proof might be necessary for it to make a Better Bug Proof. Viruses "infect", and therefore leave some trace in, readable storage. Bugs can show up as volatile output, such as to screen displays. Bugs can also show up in non-machine-readable forms, such as the output of printers. To include bugs which show up only on screens and in printouts, I can broaden the output definition. One way would be to include a trace of all writes to unreadable or volatile media.

■ Questions about specific scripts

It's time to look at questions that Rice's Theorem does not apply to. For Rice's theorem to apply, the question has to be about partial functions in general. If I'm only asking about some partial functions or some Perl scripts, then Rice's Theorem does not apply, at least not directly.

For example, above I proved that it is undecidable in the general case whether a Perl script prints the digit "0". Does that mean that I can't decide that the Perl script

```
say 0;
```

prints 0? Not at all. Similarly,

```
say 42;
```

does not print 0 and I can decide that.

Rice's Theorem does not apply to questions about specific Perl scripts. Rice's Theorem also does not apply to questions about finite sets of Perl scripts. Rice's Theorem only directly applies to questions that are about all functions performed by Perl scripts.

While, pedantically speaking, Rice's Theorem can't be used to answer questions about subclasses of Perl scripts, logic hacks can work around that restriction. If I use logical connectives (and's, or's and not's), and multiple questions, I can formulate questions about all Perl scripts that imply the answers to questions that are only about subclasses of Perl scripts.

I can indirectly apply Rice's Theorem to any question about any subclass of Perl scripts, if

- The subclass is defined by a partial function.
- The subclass is non-trivial. (There is at least at least one Perl script and one input not in the subclass.)
- The question is about a partial function.
- The question is non-trivial for that subclass. (There's a Perl script and an input in the subclass for which the answer is yes, and another Perl script and an input in the subclass for which the answer is no.)

For example, pedantically speaking, in Rice's Theorem, I can't restrict the question about writing to STDERR only to scripts which open sockets. But I can ask if it is decidable whether a script which opens a socket, also writes on STDERR. (Using logical connectives and pseudo-code, this would be `NOT` `opens_socket OR writes_stderr`.) I can also ask if it is decidable whether a script which opens a socket does not write on STDERR. (`NOT opens_socket OR NOT writes_stderr`.) Rice's Theorem applies to both these questions, and they are undecidable. This tells me that the question of whether Perl scripts write to STDERR is undecidable, even if I am limiting consideration to Perl scripts which open sockets.

■ Is versus does: the litmus test----------------------------

In many cases it's clear what is meant by the difference between what a script "does" (the partial function it implements) and what a script "is" (properties of the script which are not properties of the partial function.) But not always. For example, take the question "Is the script recursive?"

This is an "is" question, and Rice's Theorem does not apply. Intuitively, this might seem like a "does" question and it might seem that Rice's Theorem should apply.

Fortunately, there's a litmus test. If any two Perl scripts which implement the same function have different answers to a question, then that question is an "is" question, and Rice's Theorem does not apply. Otherwise, it's a "does" question, and Rice's Theorem does apply.

If I can find a function which has both a recursive solution and a non-recursive solution, that will be enough to show that the question "Is the script recursive?" is one of those not addressed by Rice's Theorem. Many problems have both recursive and non-recursive solutions. Rather than use an elegant pair, I'll settle for a simple example. The code below copies a single line from `STDIN` to `STDOUT`. Nobody in their right mind would solve this problem recursively, especially in Perl. But here it is:

```
sub inefficient {
   return unless @_;
   print (shift);
   inefficient(@_)
   }

inefficient(split //, <STDIN>);
```

This same function is implemented much more nicely by the following non-recursive script:

```
     print;
```

This demonstrates that the question of whether or not a Perl script is recursive is not about partial functions, and therefore is not a question whose decidability Rice's Theorem can determine.

■ Decidable: does it run longer than N seconds? -----

If a Perl script implementing a partial function runs in less than N seconds, I can write a slower one that implements the same partial function. I can write it less efficiently, or I can just insert pointless logic. Since I can write two scripts which perform the same partial function, but have different answers to the question, clearly the question of run-time length is not about the partial function. Rice's Theorem does not apply.

Pedantically, If N is less than Perl's start-up time, Rice's Theorem does apply, because every script produces the same answer. That means every script for every partial function produces the same answer, and technically speaking, the question is about partial functions. But in that case the answer to the question is always "yes", so the question is trivial. Rice's Theorem applies to trivial questions, but it does not prove undecidability for them.

I can easily decide whether a Perl script takes N seconds to run or not. I start the script and time it. But for one question about about how long a Perl script runs, Rice's does prove undecidability. That's the question of whether a Perl script runs forever -- the Halting Question.

For the Halting Question, all scripts implementing the same partial function have the same answer. When the question is whether the program will run forever, making the script less efficient doesn't change the answer. Timing a Perl script doesn't help. Timings cannot reliably tell the difference between Perl scripts which run forever, and Perl scripts which halt after running for a very long time. For the Halting Question, Rice's Theorem applies and proves undecidability.

■ Proof: Perl is unparseable ---------------------------------

This proof depends on a parsing ambiguity illustrated in the following elegant example, which is taken from a posting by ikegami on Perlmonks.

```
$ perl -E 'sub dunno { 3 } say dunno + 4'
3
$ perl -E 'sub dunno() { 3 } say dunno + 4'
7
```

In both commands the `dunno` subroutine returns 3, ignoring any argument it is passed. In the first, there is no prototype, so that "`dunno + 4`" is parsed as a call to `dunno` with `+4` as its argument. The `+4` is discarded and the command prints "3".

In the second command, there is a nullary prototype, and the plus sign in "`dunno + 4`" is parsed as a binary operator. `dunno` is passed no arguments and the 3 which it returns is added to the 4. The result is "7".

This proof will proceed by showing that it's undecidable whether `dunno` has no prototype, as in the first command, or a nullary prototype, as in the second. Without knowing how `dunno` is prototyped, I can't know how the lines of Perl in the example above are parsed. Since `dunno`'s prototype is undecidable, the parses of the lines above are undecidable. Since the parses of the lines above are undecidable, Perl parsing in general is undecidable.

Input: Not relevant.

Output: A printout showing how some Perl test code would be parsed, when preceded by the Perl code in question. This could be obtained by concatenating the test code to the

Perl code in question, running the combined code with the `-MOConcise,-terse` flags, then extracting the parse for the test code from the output.

Question: For any Perl code, any input, and the test code "`say dunno + 4`", does the plus sign in the test code parse as a binary operator?

"No" case:

```
 BEGIN { *dunno = sub { 3 } }
```

"Yes" case:

```
 BEGIN { *dunno = sub () { 3 } }
```

The question is about the output of constant functions, and is non-trival. So the parse of the test code is undecidable. QED.

There's a special requirement in this proof. The output in this proof is a parse, and a parse is produced before run-time. I must show that I can use Turing-complete Perl code to determine the prototype of the `dunno` subroutine at compile time.

A function definition won't work. Function definitions take effect before the execution of any Perl code, even the code in `BEGIN` blocks. For Turing-complete Perl to choose `dunno`'s prototype, I need to establish the prototype before the test code is compiled, but I cannot use a function definition.

In the non-triviality conditions, I set up prototypes for anonymous subroutines. I give the anonymous subroutine of my choice the name `dunno` using symbol table manipulation. I do this in a BEGIN block, it is available at compile time, and it affects the parse of the test code.

Larry Wall says there is more than one way to do it. Even a ragged-edge hack like this is no exception. Another way to set up the prototypes is to put function definitions into strings. The strings can be `eval`'ed in a `BEGIN` block. The `eval`'ed function definitions will be available when the test code is compiled. Their prototypes will affect the parse.

■ How Rice's Theorem is proved---------------------------

I won't prove Rice's Theorem here. It is proved twice in the Wikipedia article on Rice's Theorem: once informally and once with some rigor. The Rice's proof is very similar to the one for the Halting Theorem, which I gave in Perl-ish form in the first article of this series. In the next article in this series, I will give another proof that Perl is unparseable. That proof that will follow the same strategy as the two Wikipedia proofs of Rice's Theorem.

The "Formal Statement" in the Wikipedia article on Rice's Theorem describes the partial functions using integers as their input and output. I use strings to represent input and output. So do the proofs in the Wikipedia article. For an actual implementation, strings would be far superior. But integers have been standard in math.

Integer and string representations of input and output are equivalent. Each can be mapped to the other in many ways. One way to map every integer to a string is after the fashion of `Math::BigInt::bstr()`.

At machine level, every string is already represented as a number. I could implement the mapping of an arbitrary length string to an integer by taking the numerical value of each character of a string and using `Math::Bigint` to do arbitrary precision shifts and adds. That would still fail to capture the full theoretical concept, since the Perl implementation won't have infinite memory available to it. But it's not necessary to indicate how to write these mappings in Perl. All that is needed is to show that mappings exist.

The "Formal Statement" in the Wikipedia article on Rice's Theorem also refers to Gödel-encoding. A Gödel encoding is a way of representing partial functions as integers. Every partial function has at least one Perl script that represents it, and this is an easy and efficient Gödel encoding of partial functions to strings. For a Gödel encoding to integers, the Perl scripts can in turn be mapped to integers, just like any other strings.

■ Conclusion ---

Rice's Theorem is flexible and has wide applications. Results come easily. So easily that Rice's can seem like a "black box". Out pops the answer, but sometimes no feeling for why the proof is true pops out along with it.

In the next and last article in this series, I will go back to basics. I will prove the Perl unparseability result without invoking Rice's Theorem. This proof will be similar to that

given for the Halting Theorem in my first article, and will bring us full circle.

■ References--

William Dowling's Virus Proof: "There Are No Safe Virus Tests", William Dowling, *American Mathematical Monthly*, v.96 n.9, p.835-836, Nov. 1989. ISSN 0002-9890. *http://vx.netlux.org/lib/awd00.html*

Rice's Theorem: *http://en.wikipedia.org/wiki/Rice%27s_Theorem*

Turing Completeness: *http://en.wikipedia.org/wiki/Turing_completeness*

ikegami's elegant example of parsing ambiguity in Perl, from Perlmonks: *http://perlmonks.org/?node_id=688260*

"Perl Cannot Be Parsed: A Formal Proof", my original Perlmonks post on Perl's unparseability: *http://www.perlmonks.org/?node_id=663393*

My general BNF parser, currently in alpha: *http://search.cpan.org/dist/Parse-Marpa*

Adam Kennedy's PPI module: *http://search.cpan.org/dist/PPI*

■ About the author ---

Jeffrey Kegler has been using Perl since 1987. At the time, he'd bid a fixed-price gig and took a chance that the newly-released Perl 1 would be better than shell scripting. Betting on Larry Wall proved to be a good move. Jeffrey is the author of the `Test::Weaken` and `Parse::Marpa` CPAN modules.

Jeffrey is a published mathematician, has a BS and an MSCS from Yale, and was a Lecturer in the Yale Medical School. In 2007 he published his first novel, *The God Proof*. It centers on a little known part of Kurt Gödel's work—his effort to prove God's existence. Gödel worked out his proof in two notebooks: notebooks which were missing from his effects when they were cataloged after this death. *The God Proof* begins with Gödel's lost notebooks reappearing in a coastal town in modern California. It's available as a free download: *http://www.lulu.com/content/933192*. You can purchase print copies at Amazon: *http://www.amazon.com/God-Proof-Jeffrey-Kegler/dp/1434807355*.