Perl And Undecidability

The Halting Problem

by Jeffrey Kegler
jeffreykegler@mac.com

How do I know what a Perl program is going to do without running it? Does it matter that it's Perl instead of some other language? It turns out that the answer is related to the Halting Problem, which says that there is no *general* solution to the question of whether an arbitrary computer program will ever stop running. If I can't decide that, I can't perfectly analyze source code to figure out what it is doing without running it.

I first started thinking about this problem because it's a thorn-in-the-side for Adam Kennedy's PPI module, which attempts to parse Perl code without running it. PPI is the backbone of `Perl::Critic`, a tool to enforce code policy. Can PPI ever be 100% correct? I won't answer that question in this article, but I will go through the Halting Problem to show you how it applies to Perl. This is a first in a series of three articles that will eventually give you that answer.

These articles will avoid mathematical notation in favor of Perl 5. Perl 5 has its disadvantages for this work but it is an excellent way of presenting algorithms, in many ways much less awkward than the mathematical notations. I hope some readers, seeing the ideas in this form, will become comfortable enough with them to tackle them in more standard notation. If you decide you want to do that, Wikipedia is an excellent place to start.

■ Undecidability

A computer problem is *undecidable* if it's a decision about which of two sets an input falls into, and it's impossible to make that decision. Every computer problem can be seen as one or more decisions between two choices, so that all limits to decidability are limits to what can be done by computer. This is the first in a series of three articles about an insight that undecidability gives us into Perl, and the insights Perl gives us into undecidability.

These days computability and undecidability are treated as synonyms, but questions about decidability predate our modern notion of computing. The first person to formalize computability was Alan Turing, who introduced the Halting Problem. The Halting Problem is the problem of deciding whether a given computer with a given input runs forever or eventually halts. Turing didn't call it the the Halting Problem, but since his day it's gotten that name. Turing showed that the Halting Problem was uncomputable and therefore undecidable.

The Halting Problem is far from the only undecidable problem. There are many others, and it is not at all hard to run into them in everyday practice. It is usually obvious that an undecidable problem is going to be, at the least, very difficult. You often avoid them anyway, and in that case whether a problem is undecidable or merely just plain hard is a quibble. But there are also problems whose solution, if possible, justifies major resources. Knowing a problem is undecidable allows you to either redefine it or give up.

■ Borrow your cat?

There's a big problem with writing a proof in Perl. When I see Perl code, it raises a set of expectations. Most of those expectations are wrong when it comes to proofs. Perl code is usually written to *do* something. This code with this article is not: it's written to explore an idea as a thought problem.

The most famous example of a thought problem is Schrödinger's cat. In it, you need a cat. You lock him in a box shielded not only from outside interference, but from outside observation. Inside this box you put another box. This inner box is not airtight, but it is shielded from interference by the cat. The inner box has a geiger counter triggered to detect atomic decays. The geiger counter has a relay which controls a hammer hanging over a flask, closed, fragile, and filled with a gas of high feline toxicity. Dr. Schrödinger suggested hydrocyanic acid.

According to quantum mechanics, the geiger counter, as long as it is not observed, never trips the relay, never releases the hammer, never breaks the glass, and never kills the cat. But neither is the opposite true. You cannot say that the relay was tripped, that the hammer dropped, that the flask was broken, or that the cat is dead. The situation inside the box is a mixture of the two, not even a probability, but a kind of probability wannabe called a quantum superposition. When the box is opened the superposition collapses into a probability, the probability collapses into a fact, and the cat either either collapses dead or springs out of the box.

So what's up with the cat before the box is opened? That's the point of the problem. It doesn't really engage our intuition to be told that the decay state of an atom is a mathematical function involving complex numbers. To be told that about the trace a geiger counter leaves on its recording tape is a bit more of a challenge. But neither compares to the challenge involved in trying to envision a cat who is not alive, not dead, not in a transition from life to death, not in a transition from death to life, and not even in a state which is a red-blooded real-numbered probability of life or death. Now that's a challenge to our intuition.

If you think about Schrödinger's cat like as an engineer, you will ask questions like: If the box is airtight, how does the cat breathe? If the box is not airtight, does the cat's respiration interact with the outside world? Doesn't that ruin the experiment? How do you calibrate the geiger counter? Since hydrocyanic

acid remains liquid at room temperature, is it really the right choice? Isn't the bit with the hammer and the glass flask a bit Rube Goldbergish? Shouldn't you avoid having to deal with broken glass? And what about the SPCA? Asking any of these questions is a sign that you've missed the point.

A thought problem can have major implementation issues and still work as a thought problem, as long as in principle it could be done. In fact, the difficulties from a practical point of view could be insuperable, and the thought problem will be none the worse. A thought problem can be also be far-fetched, grotesque and a very unreasonable thing to do.

■ The Heath Robinson Programming Foundation-----

Let's suppose the Heath Robinson Programming Foundation (HRPF) recognizes Perl as its official language and decides to put its vast financial resources to work on improving what's available on CPAN. The first thing it decides it needs is an infinite loop detector.

The HRPF decides to hold a contest, with a handsome prize to be awarded to the best implementation of such a detector. The HRPF board wants the submissions culled before it picks finalists. I'm offered an insultingly small grant to check that the infinite loop detectors work as specified in the contest rules. The times being what they are, I decide to take the offer.

I've reserved `Acme::Halt` on the Comprehensive Perl Archive Network (CPAN). The rules require that the `Acme::Halt` module contain a function named `halt`. The first argument to `halt` is a code reference. It's the required argument. The remaining arguments are optional, and are treated as arguments to use when `halt` dereferences the code reference.

The function referenced by the first argument, with the arguments to `Acme::Halt::halt` passed on to it, is the *test case*. For example, when the call to `Acme::Halt::halt` is:

```
Acme::Halt::halt(\&my_function,

   42, 91, \*STDERR)
```

then the test case is `my_function` with the rest of the arguments:

```
my_function(42, 91, \*STDERR)
```

`halt` returns 1 if the test case halts, and it returns 0 if the test case runs forever. There's no requirement that `halt` actually run its test case. It may run it partially or not at all. There's no formal ban on `halt` running the test case completely, but `halt` must always return a result, and it can't do that if it tries to run an infinite loop to the end. At least sometimes, `halt` must analyze its test case without running it to completion.

The test case has to follow these rules. It must:

- have no side effects
- be completely deterministic
- have no dependency on outside data other than its arguments

I might object that these restrictions turn the hard problem of finding infinite loops into three other problems, each one of which is even harder than the original problem. Bear in mind two things: a thought problem does not have to be a reasonable thing for anyone to want to do, and the HRPF has a lot of money to hand out.

■ On to work ----------

I'm expected to automate the process of checking entrants to the HRPF Infinite Loop Detector Contest. To get the first subpittance of my grant I need to submit my test plan with my Perl code. The result of my efforts is the code in *Listing 1*. The test case it uses is the function I have imaginatively named `test_case`, called with a reference to itself as its only argument.

My code doesn't seem to work right. I've already run through the money I expect from the grant, and I've got to submit something if I'm to have any hope of paying the bills when they come.

Making my best attempt, I analyze the behavior of my test program. *Table 1* shows what I find.

`halt()` returns:	What `halt()` reports (line 7)	What actually happens (line 26)
0	loops	halts
1	halts	loops

Table 1: Behavior of `test_case(\&test_case)`

In other words, whatever `Acme::Halt::halt(\&test_case, \&test_case)` says that `test_case(\&test_case)` is doing, it is actually doing the opposite. When the result of test says it should halt, the actual test case loops forever. When the result of test says it should loop forever, the actual test case halts.

I work out the details for the 91st time. Here's what I see.

Acme::Halt::halt returns 0

Line 1 includes the `Acme::Halt` package. My test case, calling a routine named `test_case` with a reference to itself as its only argument, occurs at line 26.

`test_case` shifts off its argument (line 6), which is a code reference to itself. `test_case` then calls `Acme::Halt::halt` (line 7), passing its own argument on twice. The call to `Acme::Halt::halt` at line 7 therefore amounts to a direct call to `halt` using the same reference as the first and second arguments:

```
Acme::Halt::halt(\&test_case, \&test_case)
```

What does `Acme::Halt::halt` return? I'm going to analyze both cases. I'll start with the simplest. Suppose `Acme::Halt::halt` returns 0.

Listing 1: Test program for the HRPF Infinite Loop Detector Contest

```
 1  use Acme::Halt;
 2
 3  # A routine to bust the fraudulent claimants
 4  sub test_case {
 5
 6     my $arg = shift;
 7     return 0 if Acme::Halt::halt($arg, $arg) == 0;
 8
 9     # I don't reach this point unless there's no infinite
10     # loop according to Acme::Halt::halt
11
12     # loop forever
13     my $threshold = 1024;
14     while (++$i) {
15         if ($i > $threshold) {
16             $threshold *= 2;
17             warn q{if there's no infinite loop, why haven't I finished?};
18         }
19     }
20
21     # NOTREACHED
22
23  }
24
25  # What does this routine do, if Acme::Halt::halt detects infinite loops?
26  if (not test_case(\&test_case)) {
27      die('non-existent infinite loop reported by Acme::Halt::halt()');
28  }
29
30  # NOTREACHED
```

As noted in Table 1, If `Acme::Halt::halt(\&test_case, \&test_case)` returns 0, `Acme::Halt::halt` is saying that `test_case` calling itself does not halt; it runs forever.

As we can see from line 7, the 0 return from `Acme::Halt::halt` also means we that we will return out of `test_case`. And that means we execute this `die` statement at line 27:

```
die('non-existent infinite loop reported by
Acme::Halt::halt()');
```

As my error message reports, there's a problem. At line 7 `Acme::Halt::halt` reported that `test_case(\&test_case, \&test_case)` runs forever. But at line 26 it did not run forever. As noted in *Table 1*, what is reported at line 7 contradicts what happens at line 26.

What to do about it? I decide to defer my decision until I've run through `test_case` again, this time seeing what happens if `Acme::Halt::halt` returns 1.

Acme::Halt::halt returns 1

We run `test_case` again. Nothing is different until we hit line 7. We looked at what happens if `Acme::Halt::halt` returns 0, this time let's look at what happens if the return value of `Acme::Halt::halt` is 1. Here goes.

First off, if `Acme::Halt::halt(\&test_case, \&test_case)` returns 1, that means line 7 is saying that `test_case(\&test_case)` halts. In other words `test_case(\&test_case)` does **not** run forever.

A return of 1 from `Acme::Halt::halt` also means the `return` on line 7 does not get executed and I fall through to line 13. As I can easily see, lines 13 through 19 are an infinite loop. Line 21 will never be reached. `test_case(\&test_case)` will never return.

Line 30 will also never be reached. In fact, I'll never leave the `test_case(\&test_case)` call in line 26.

The situation is as noted in Table 1. At line 26 `test_case(\&test_case)` loops forever. At line 7, `Acme::Halt::halt` says that `test_case(\&test_case)`

does not loop forever. Again, lines 7 and 26 contradict each other. As a test program, my code is a failure. I wonder how to fix it. I'm awake, watching early morning TV, when an idea occurs to me.

■ Desperate men do desperate things ----------

Mathematicians often turn statements of non-existence into statement about existence and work backwards. They show the existence of something creates a contradiction. That contradiction proves the thing never really existed.

In the pure form in which mathematicians do this kind of reasoning. It seems backwards, but we use this sort of logic all the time. Take the guy with the sure-fire investment scheme on TV. If it was that good, we say, he wouldn't be buying TV time to tell people. He'd tell a few people at most, invest his and their money, and get rich, all the time going to great lengths to keep the secret. It's got to be a scam, we say. Our reasoning went like this:

- ***Assume A***: the scheme in the infomercial is a sure-fire money maker.
- ***If A, then B***: if it was a sure-fire money-making scheme, the people who knew about it would keep it a secret.
- ***Not B***: they are not keeping it a secret.
- ***Therefore, not A***: It is not a sure-fire money-making scheme.

This pattern occurs a lot in proofs:

Assume A.
If A, then B.
Not B.
Therefore, not A.

That's called a reduction to absurdity and math would be impossible without it. I assume something, then create a contradiction. Using that contradiction, I prove that the thing I assumed is not right.

In my late night TV meditation, Assumption A is that I'm watching a sure-fire money making scheme. Contradiction B is the observation that people keep sure-fire money-making schemes secret.

■ My problem solved ----------

I'm laying back, basking in my immunity to the lure of late night scams when the solution to my halting investigations occurs to me.

I've checked my HRPF Contest Entry test program a thousand times, so I know it all works, and that my test case satisfies the restrictions. But, I realize, there is one part of it that I do not know works. I don't know that the `Acme::Halt::halt` program works as assumed.

So the situation comes down to: if (A) `Acme::Halt::halt` finds infinite loops, then (B) my test program detects them in my test case. But I also know: (Not B) My test program does not detect infinite loops in my test case.

I realize that if I make "`Acme::Halt::halt` finds infinite loops" my Assumption A, and "test program detects loops in my test case" my Contradiction B, and plug these into the formula for a reduction to absurdity, then I get this:

- ***Assume A***: Assume that `Acme::Halt::halt` finds infinite loops.
- ***If A, then B***: If `Acme::Halt::halt` finds infinite loops, the test program finds them in its test case, `test_case(\&test_case)`.
- ***Not B***: Table 1 shows the the test program will not find infinite loops in the test case.
- ***Therefore, not A***: `Acme::Halt::halt` does not find infinite loops.

In this reasoning, the only assumption I made about `Acme::Halt::halt`, was that it obeys the restrictions imposed by the contest. So the logic above is valid for all possible contest entries—I don't even have to actually test them. No valid contest entry could possibly qualify as a finalist.

■ My letter to the HRPF ----------

In drafting up my letter to the HRPF telling them that there cannot possibly be a winning entrant to their contest, I point out that they could revise the contest. For example, it is obvious that a Perl script to simply output a message does not run forever:

```
say "goodnight, Gracie"
```

I could write a program that figures that out. A program could also figure out this loops forever:

```
for (;;) { say "I'm bored." }
```

Programs to detect infinite loops can exist. All I've proved is that the general problem of finding infinite loops cannot be solved. It might be possible to write a module that finds a lot of infinite loops. A module that detects 90%, or even 50%, of all infinite loops might still be useful. It's just that such a module will have to have some kind of failure rate. Perhaps it won't detect all infinite loops. Perhaps it will detect all infinite loops, but at the price of reporting false positives—infinite loops in code which doesn't have them.

I go on, in my letter to the HRPF, to say that actually, a determination that a contest is unwinnable amounts to a winning entry. So in addition to the pittance I was promised for qualifying entrants to the HRPF Infinite Loop Detector Contest I'd like the handsome sum they promised as first prize.

■ Laying it on thicker ----------

The HRPF Infinite Loop Contest entries did not have to work for arbitrary Perl functions. Those with side effects, those with dependences on the outside other than their arguments, and those which were non-deterministic, were excluded. What happens to the proof if they are considered?

All those factors were excluded because they made the infinite loop problem even harder to decide. Since the infinite loop problem is undecidable even without those functions in the mix, I can be sure it's not decidable if their behavior needs to be decided as well.

In fact, I tell the HRPF board that these restrictions forced the proof to be especially strong—forced it to show that undecidability is not limited to situations which involve randomness, side effects, or outside dependencies. It shows their brilliance in framing the contest the way they did. On the same page, I include the address to which they can send the check for first prize.

Does the HRPF, overwhelmed with pride and gratitude, give me First Prize? I'll leave that to your imagination. Thought problems can be far-fetched, but there are limits to everything.

■ In defense of the halting problem ----------

Mathematicians almost always prove a problem is undecidable by reducing it to the Halting Problem. The Halting Problem has tradition behind it. and infinite loops relate reasonably well to real-life computing. But mathematicians often pretend they don't care about real life. They also use those horribly klunky Turing machines, with their read heads and squares marked on infinite length tapes. Is it because Theory of Computation professors are elitist jerks, focused on irrelevancies? That is certainly true of many of them, but it's not the real story.

When Turing formulated the Halting Problem, electronic computers did not exist. The term "computer" usually referred to a woman. A lot has changed since then. But do we really understand the central issues of our field so much better than Turing? Fifty years from now our models of computing will look as klunky as Turing's tapes and read heads. When it comes to thinking out the issues that will remain issues over the decades, current technology is not clearly superior.

Turing knew his physical technology could be superseded, something some of us perhaps forget about our own technology. Turing knew he was building ideas, and he built ideas that have lasted. Until we need a model of computing for which Turing machines and the Halting Problem aren't adequate, why change?

■ Conclusion ----------

In this article I presented a proof of the Halting Problem using Perl as its notation. It required looking at Perl in a different way—as a framework for a thought problem, rather than as a way of getting a job done. In particular, the code I presented didn't accomplish its goal within the thought problem, and it called a function whose existence was doubtful. The proof turned on showing that everything else about the code worked fine, so that the assumption that the function existed was the weak link. Since the only thing I'd assumed about the function was that it was callable from Perl and that it solved the Halting Problem, no Perl function to solve the Halting Problem can exist.

This means the Halting Problem is undecidable in the Perl context. In fact it's undecidable for any modern general-purpose computer language. The Halting Problem is not the only undecidable problem. You may be surprised to learn how many useful questions have undecidable answers. The second article in this series will present the extremely useful Rice's Theorem, a fast, easy way of spotting undecidable problems. Rice's Theorem is not well-known among working programmers, but it should be.

As I said at the beginning, I began thinking about this issue while reading Adam Kennedy's `PPI` module. I was looking at a potential application for a parser generator. Parsing Perl 5 looked like it might be a challenge. I ran across Adam's conjecture that Perl 5 parses were in fact undecidable, and his hint at how to prove that. Any number of people in the Perl community know enough math to have formalized the proof that Adam outlined. Of this number, I was the first to be too dim to see immediately that Adam was right. The third and last article will contain that formal proof, again in Perl 5 notation.

■ References----------

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

The Halting Problem:
http://en.wikipedia.org/wiki/Halting_problem

Reduction to absurdity:
http://en.wikipedia.org/wiki/Reductio_ad_absurdum

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 the of `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*.