When have you come upon the halting problem in the field? [closed]

Asked 16 years, 2 months ago Modified 6 years, 1 month ago

Viewed 6k times

67

Closed. This question is opinion-based. It is not currently accepting answers.

Want to improve this question? Update the question so it can be answered with facts and citations by editing this post.

Closed 10 years ago.

Improve this question

When have you ever personally come upon the **halting problem** in the field? This can be when a co-worker / boss suggested a solution which would violate the fundamental limits of computation, or when you realized yourself that a problem you were trying to solve was, in fact, impossible to solve.

The most recent time I came up with it was when studying type checkers. Our class realized that it would be impossible to write a perfect type checker (one that would accept all programs that would run without type errors, and reject all programs that would run with type

errors) because this would, in fact, solve the halting problem. Another was when we realized, in the same class, that it would be impossible to determine whether a division would ever occur by zero, in the type-checking stage, because checking whether a number, at run-time, is zero, is also a version of the halting problem.

language-agnostic field theory turing-machines halting-problem

Share

Improve this question

Follow

edited Nov 6, 2018 at 15:38

Cœur

38.6k ● 26 ● 202 ● 276

asked Oct 25, 2008 at 6:08

Claudiu

229k ● 173 ● 503 ● 697

1 don't static type systems check only for the type of the variable instead of its value? I think your class malformed the question when expecting a static type checker to reject runtime errors at compile time. – andandandand Jul 5, 2009 at 14:21

1 @dmindreader - No. Most compilers/type-safe languages indeed just check for types, but it's possible to see the range of values for something (sometimes) given static analysis. Consider how ReSharper or Coverity produce the "possible null value" warnings. – Robert Fraser Jan 6, 2010 at 19:35

3 I used to design medical devices. I was once asked to include, in a battery powered device, a light that would

indicate that the battery was dead. – David Schwartz Dec 25, 2011 at 19:27

1 @DavidSchwartz: A battery that is too dead to power the device can still power an LED for long enough to be noticed by someone who can replace the battery. – endolith Jan 22, 2013 at 20:27

13 Answers

Sorted by: Highest score (default)

61

I *literally* got assigned the halting problem, as in "write a monitor plugin to determine whether a host is permanently down". Seriously? OK, so I'll just give it a threshold. "No, because it might come back up afterward."

Much theoretical exposition ensued.

Share Improve this answer

Follow

answered Oct 25, 2008 at 6:13

Kirk Strauser

30.9k ● 5 ● 52 ● 69

5 Wow - just, wow. The depths of the illogic that must exist in that person's mind just baffle me... – Erik Forbes Oct 25, 2008 at 6:27

14 This problem is actually very easy to solve in .NET. Just add a reference to System.Future, and then there are some static methods in the System.Future.Fact class you can use. In your case: if (System.Future.Fact.IsPermanent(delegate to check if server is down here)) { ... } – Lasse V. Karlsen Dec 3, 2008 at 12:49

5 @Lasse that would have been more believable if we were talking of Python :) (not that I don't love .NET, but you know... `import antigravity` and all...) – Roman Starkov Aug 13, 2010 at 23:56

7 @Erik No, *you* are the illogical one. The program is very simple: `host.self_destruct(); print("Permanently down.");` . – Mateen Ulhaq Nov 4, 2011 at 2:50

6 @muntoo you seem to have some unreachable code at row 2 – Drathier Sep 17, 2013 at 20:08

45

Years ago, I remember reading a review (in Byte magazine, I believe) of a product called Basic Infinite Loop Finder or BILF. BILF was supposed to scan your Microsoft Basic source code and find any loops which didn't terminate. It claimed to be able any find any infinite loops in the code.

The reviewer was savvy enough to point out that for that program to work in all cases, it would have to solve the halting problem and went so far as to provide a mathematical proof of why it couldn't work in all cases.

In the next issue, they published a letter from a company representative explaining that the problem would be fixed in the next release.

Update: I ran across an image of the article on imgur. I remembered the wrong magazine. It was Creative Computing, not Byte. Otherwise, it's pretty much as I remembered it.

You can see a hi-res version of it on imgur.

BILF: BASIC Infinite Loop Finder

An objective review of one of the most useful software packages on the market!

Jeff Levinsky

This article is intended to warn those who have contemplated purchasing the BILF (BASIC Infinite Loop Finder) program offered for BASIC users by the Cosmic Software Corporation in last month's issue of *Popular Megabyte*. BILF, as I have painfully discovered, and its soon-to-be-released big brother (known as Super BILF), are little more than impressive hoaxes: the infinite loop finders must and do fail.

Actually, the idea behind BILF is a good one. Infinite loops, as we all know, are sections of code that are inescapable once entered (a software analogy of a black hole). The simplest possible infinite loop in BASIC is:

```
10 GO TO 10
```

which is of course inescapable. However, large BASIC programs may contain far more insidious infinite loops which are almost impossible to detect. BILF claims to be able to detect any infinite loop no matter how concealed.

My own motivation for purchasing BILF was to check some large puzzle solving programs that I have written. These puzzle programs examine all possibilities at each stage in the solution of the puzzle and are inherently very slow. To solve the puzzles for larger and larger boards, the time required becomes much greater. For example, my 8080-based computer requires only 30 seconds to produce a winning strategy for one puzzle on a three by three board. However, I have calculated that as much as 27 hours would be needed to solve the very same puzzle for a four by four board. But what if I did not know how long the computer would take to solve the puzzle and the program contained an infinite loop? I would run the program, see no results, but then assume that perhaps in another moment the program would halt, wait out that moment, make the same assumption, wait out another moment, etc., etc. I would never be sure whether or not the program had failed. With BILF, I can obviously analyze my puzzle solvers and then be confident that they will not infinite loop but instead will eventually halt.

Now for the actual details on BILF. Cosmic Software charges 20 dollars for it (Super BILF will cost 30 dollars) and supplies it on a standard cassette which loaded easily into my machine. BILF is written in standard BASIC and consists of 260 lines of the most obscure code imaginable. Once loaded, BILF is set to contain the address of the program to be tested *and* the address of the data that the (test) program is to be run with. Both the test program and the test data must be in memory. The reason that BILF requires the latter is that a program may infinite loop only upon certain input. The program below will infinite loop if, and only

```
10 INPUT I
20 IF I=0 THEN 20
30 PRINT "MADE IT!"
40 STOP
```

if, the inputted number is a zero. So, in order for BILF to decide whether or not this program will infinite loop, BILF must know what number will be inputted, that is, the data. Figure 1 gives a pictorial summary of this.

I first used BILF to test some very simple programs, such as the one above. In all cases, BILF worked admirably: it quickly and consistently produced the correct answer. I then tested my puzzle games. As a testament to my computer skills, BILF found all of my programs to be free of infinite loops. In short order, I had tested all the programs that I had, save BILF itself. Testing BILF on itself presented something of a challenge, but I was curious to know if BILF might somehow infinite loop, thereby failing to decide whether or not its test program would infinite loop. Although this possibility sounds bizzare, it was easy to try: I set the address of the test program for BILF to be that of BILF itself. The address for the test data did not matter in this instance because BILF does not contain any input statements. Using the same sort of notation as above, I ran:

Figure 1

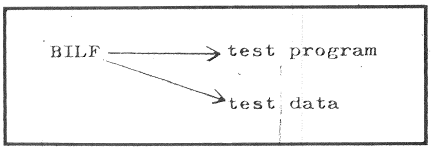
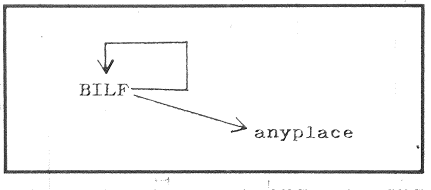


Figure 2.



As the arrow shows, the program that BILF tested was BILF itself.

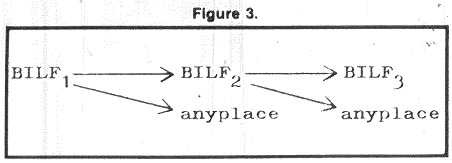
There a few tricky points here. Note that in this test of BILF, the BILF being checked for infinite loops is itself checking BILF for infinite loops. This is due to the fact that both the BILF that I am running (which I will call $BILF_1$) and the BILF that it is testing (which I will call $BILF_2$) are one and the same, and therefore share the address of the test program. So when $BILF_1$ tests $BILF_2$, $BILF_2$ tests BILF also. I shall call this third instance $BILF_3$. In the same notation as before, we have:

This is equivalent to Figure 2. I suspect that some people will insist here that $BILF_3$ must be testing a $BILF_4$ which is, in turn, testing a $BILF_5$ and so on. Whether or not this is true is immaterial. The important observation is that $BILF_1$ is testing $BILF_2$ testing $BILF_3$.

Upon running the above test, I discovered, as might be expected, that $BILF_2$ testing $BILF_3$ does not infinite loop.

Jeff Levinsky, 1697 Calle Leticia, La Jolla, CA 92037.

Figure 3.



Since all three BILFs are actually the same here, I can state that BILF testing BILF does not infinite loop. Of course, if $BILF_2$ testing $BILF_3$ *did* go into an infinite loop, so would $BILF_1$ testing $BILF_2$: my experiment would never have halted.

The next step in my study of BILF was to modify it in a very simple but devious way. The very last lines of BILF are:

```
9800 PRINT "THIS PROGRAM WILL INFINITE LOOP"
9810 STOP
9900 PRINT "THIS PROGRAM WILL NOT INFINITE
     LOOP"
9910 STOP
```

Apparently, once BILF determines that the test program will infinite loop on the given input, it will go to line 9800 to print the message there. Then BILF will stop. On the other hand, if BILF decides that the program will not infinite loop, it will go to line 9900, print the message there, and then stop. Since BILF always prints one of these two messages, I assume that BILF always ends by going to either line 9800 or line 9900. My modification to BILF was to change line 9910 into an infinite loop. This results in:

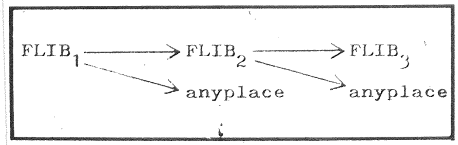
```
9800 PRINT "THIS PROGRAM WILL INFINITE LOOP"
9810 STOP
9900 PRINT "THIS PROGRAM WILL NOT INFINITE
     LOOP"
9910 GO TO 9910
```

I call this new version FLIB. With the modification, FLIB works exactly as BILF does, until after the final message is printed. Then, if the test program does not contain an infinite loop, FLIB *will* loop forever. If, however, the test program does contain an infinite loop, then FLIB will stop (and thus will *not* have an infinite loop). Only these two cases are possible, and consequently the following rules applies:

"FLIB will infinite loop if, and only if, the program it tests will not."

No doubt one question is now obvious: what happens when FLIB tests itself? This is done just as for BILF and is illustrated in Figure 4. Before indicating the actual results of running this

Figure 4.



test, I will first explain the expected outcome. Applying the rule given above for FLIB, we have:

"$FLIB_1$ will infinite loop if, and only if, the program it tests will not."

The test program is $FLIB_2$, and it is testing $FLIB_3$, so we have:

"$FLIB_1$ will infinite loop if, and only if, $FLIB_2$ testing $FLIB_3$ will not."

But all three FLIBs are actually the same, so we have:

"FLIB will infinite loop if, and only if, FLIB testing FLIB will not."

This is a paradox! To see why, consider each possibility. If $FLIB_2$ decides that $FLIB_3$ will infinite loop, then $FLIB_2$ will stop, and thus $FLIB_1$ will infinite loop. In other words, if FLIB testing FLIB stops, then FLIB testing FLIB will not. This is clearly impossible. But so is the alternative; if FLIB testing FLIB does not stop, then FLIB testing FLIB will. Thus, we have the paradox.

Since most people agree that paradoxical behaviour does not occur, FLIB cannot actually follow the rule given for it. That rule was obtained by just a simple change in BILF, and therefore BILF also cannot work as claimed. This logic leads me to believe that Cosmic Software has perpetrated an impressive but indisputable hoax. Furthermore, their Super BILF must contain the same sort of ending lines of code and thus once could then construct a Super FLIB to which the very same paradox would apply. In fact, *any* infinite loop finder cannot exist, for this very reason.

Upon realizing all of this, I wrote an irate letter to Cosmic Software, explaining in detail the arguments above and demanding a refund. As of this month, I have received only a polite reply from them stating that they are investigating the matter and have planned a new release that will fix this "bug." In my opinion, they have totally missed the point (perhaps intentionally?!) for the problem is insurmountable. I can only warn others to avoid Cosmic Software, and any other firm that makes such claims.

I have left one question unanswered. What indeed does happen when FLIB tests itself. As a matter of sheer curiosity, I have set the program running and, as of this writing, it has yet to halt. And as I see it this is exactly right. ■

Note: Although contrived, this article illustrates a valid paradox in computer science. The actual problem is typically known as the Halting Problem and was developed by Alan J. Turing, although it appears under many other names in other fields as well.

References:

Minsky, Marvin L. *Computation: Finite and Infinite Machines*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1967.

Share Improve this answer edited May 19, 2015 at 20:40

Follow

answered Jan 23, 2009 at 13:16

Ferruccio

101k ● 38 ● 229 ● 303

19 This is hilarious! Is there a copy available somewhere? – rodion Oct 16, 2009 at 16:54

1 The only results on Google are this post and links to it. – SLaks Jan 6, 2010 at 19:59

3 This would have been in the late 70's or early 80's. The web wasn't quite so popular back then ;-) – Ferruccio Jan 6, 2010 at 20:16

3 When will the BILF company stop doing business? – Geoffrey Zheng Sep 18, 2010 at 3:18

35

The project I'm working on right now has undecidable problems all over it. It's a unit test generator, so in general what it tries to accomplish is to answer the question *"what this program does"*. Which is an instance of a halting problem. Another problem that came up during development is *"are given two (testing) functions the same"*? Or even *"does the order of those two calls (assertions) matter"*?

What's interesting about this project is that, even though you can't answer those questions in **all** situations, you **can** find smart solutions that solve the problem 90% of the time, which for this domain is actually very good.

Other tools that try to reason about other code, like optimizing compilers/interpreters, static code analysis tools, even refactoring tools, are likely to hit (thus be forced to find workarounds to) the halting problem.

answered Oct 25, 2008 at 6:37

Michał Kwiatkowski
9,734 ● 2 ● 27 ● 20

Example 1. How many pages in my report?

When I was learning to program I played with making a tool to print out pretty reports from data. Obviously I wanted it to be really flexible and powerful so it would be the report generator to end all report generators.

The report definition ended up being so flexible that it was Turing complete. It could look at variables, choose between alternatives, use loops to repeat things.

I defined a built-in variable N, the number of pages in the report instance, so you could put a string that said "page n of N" on each page. I did two passes, the first one to count the pages (during which N was set to zero), and the second to actually generate them, using the N obtained from the first pass.

Sometimes the first pass would compute N, and then the second pass would generate a different number of pages (because now the non-zero N would change what the report did). I tried doing passes iteratively until the N settled down. Then I realised this was hopeless because what if it didn't settle down?

This then leads to the question, "Can I at least detect and warn the user if the iteration is never going to settle on a stable value for the number of pages their report

produces?" Fortunately by this time I had become interested in reading about Turing, Godel, computability etc. and made the connection.

Years later I noticed that MS Access sometimes prints "Page 6 of 5", which is a truly marvelous thing.

Example 2: C++ compilers

The compilation process involves expanding templates. Template definitions can be selected from multiple specialisations (good enough to serve as a "cond") and they can also be recursive. So it's a Turing complete (pure functional) meta-system, in which the template definitions are the language, the types are the values, and the compiler is really an interpreter. This was an accident.

Consequently it is not possible to examine any given C++ program and say whether the compiler could in principle terminate with a successful compilation of the program.

Compiler vendors get around this by limiting the stack depth of template recursive. You can adjust the depth in g++.

Share Improve this answer

Follow

edited Dec 4, 2008 at 10:22

answered Dec 2, 2008 at 17:27

Daniel Earwicker

117k • 38 • 208 • 286

21

Many many moons ago I was assisting a consultant for our company who was implementing a very complex rail system to move baskets of metal parts in and out of a 1500-degree blast furnace. The track itself was a fairly complex 'mini-railyard' on the shop floor that intersected itself in a couple of places. Several motorized pallets would shuttle baskets of parts around according to a schedule. It was very important that the furnace doors were open for as short a time as possible.

Since the plant was in full production, the consultant was unable to run his software in 'real time' to test his scheduling algorithms. Instead, he wrote a pretty, graphic-y simulator. As we watched virtual pallets move around on his on-screen track layout, I asked "how will you know if you have any scheduling conflicts?"

His quick answer, "Easy - the simulation will never stop."

Share Improve this answer

Follow

edited Nov 11, 2011 at 20:51

Alex Turpin

47.7k • 23 • 116 • 146

answered Nov 20, 2008 at 21:12

n8wrl
19.7k • 4 • 69 • 104

6 meta parts is why it took until the blast furnace for me to understand that his rails system was NOT a Ruby web application. – Karl Dec 2, 2008 at 17:49

1 Meta! I had to re-read my post sevral times until I understood what you meant! MetaL, of course! – n8wrl Dec 3, 2008 at 12:42

13

Sophisticated static code analysis can run into the halting problem.

For example, if a Java virtual machine can prove that a piece of code will never access an array index out-of-bounds, it can omit that check and run faster. For some code this is possible; as it gets more complex it becomes the halting problem.

Share Improve this answer

Follow

answered Oct 25, 2008 at 6:24

Jason Cohen
83k • 26 • 110 • 114

11

This is still a problem for shaders in GPU applications. If a shader has an infinite loop (or a very long calculation), should the driver (after some time limit) stop it, kill the fragment, or just let it run? For games and other commercial stuff, the former is probably what you want, but for scientific/GPU computing, the latter is what you want. Worse, some versions of windows assume that

since the graphics driver has been unresponsive for some time, it kills it, which artificially limits the computing power when doing computation on the GPU.

There's no API for a application to control how the driver should behave or set the timeout or something, and there's certainly no way for the driver to tell if your shader is going to finish or not.

I don't know if this situation has improved recently, but I'd like to know.

Share Improve this answer

Follow

answered Nov 21, 2008 at 22:59

joeld

32.8k ● 3 ● 29 ● 22

7

Another common version of this is "we need to eliminate any deadlocks in our multi-threaded code". A perfectly-reasonable request, from the management perspective, but in order to prevent deadlocks in the general case, you have to analyse every possible locking state that the software can get into, which is, no surprise, equivalent to the halting problem.

There are ways to partially "solve" deadlocks in a complex system by imposing another layer on top of the locking (like a defined order of acquisition), but these methods are not always applicable.

Why this is equivalent to the halting problem:

Imagine you have two locks, A and B, and two threads, X and Y. If thread X has lock A, and wants lock B also, and thread Y has lock B and wants A too, then you have a deadlock.

If both X and Y have access to both A and B, then the only way to ensure that you never get into the bad state is to determine all of the possible paths that each thread can take through the code, and the order in which they can acquire and hold locks in all those cases. Then you determine whether the two threads can ever acquire more than one lock in a different order.

But, determining all of the possible paths that each thread can take through the code is (in the general case) equivalent to the halting problem.

edited Oct 27, 2010 at 5:00

answered Nov 20, 2008 at 20:55

Mark Bessey
19.8k ● 4 ● 49 ● 69

1 Why is it impossible? Of course, the performance can degrade with excessive locking but that doesn't make it a halting problem! Or am I missing something here?
– Jaywalker Oct 26, 2010 at 12:23

Perl's testing system maintains a test counter. You either put the number of tests you're going to run at the top of

5

the program, or you declare that you're not going to track it. This guard against your test exiting prematurely, but there are other guards so it's not all that important.

Every once in a while somebody tries to write a program to count the number of tests for you. This is, of course, defeated by a simple loop. They plow ahead anyway, doing more and more elaborate tricks to try and detect loops and guess how many iterations there will be and solve the halting problem. Usually they declare that it just has to be "good enough".

Here's a particularly elaborate example.

Share Improve this answer

Follow

answered Oct 25, 2008 at 8:22

Schwern

164k ● 27 ● 218 ● 362

2

I found a Berkeley paper: *Looper: Lightweight Detection of Infinite Loops at Runtime*
http://www.eecs.berkeley.edu/~jburnim/pubs/BurnimJalbertStergiouSen-ASE09.pdf

LOOPER may be useful since most infinite loops are trivial errors. However, this paper **doesn't even mention the halting problem!**

What do they say about their limitations?

> [LOOPER] typically cannot reason about loops where non-termination depends on the

> particulars of heap mutation in every loop iteration. This is because our symbolic execution is conservative in concretizing pointers, and our symbolic reasoning insufficiently powerful. **We believe combining our techniques with shape analysis and more powerful invariant generation and proving would be valuable future work.**

In other words, "the problem will be fixed in the next release".

Share Improve this answer

Follow

answered Sep 16, 2012 at 16:14

Kevin Kostlan

3,509 ● 7 ● 30 ● 36

Runtime loop detection where the state space is known and fixed is actually decidable; see cs.stackexchange.com/a/11646/2242 . – Andrew Jun 4, 2016 at 15:25

1

I was once working on an integration project in the ATM (Automated Teller Machines) domain. The client requested me to generate a report from my system for transactions sent by the country switch which were not received by my system!!

Share Improve this answer

Follow

answered Nov 26, 2008 at 8:36

Jaywalker

3,109 ● 3 ● 29 ● 45

0

From the Functional Overview of (Eclipse) Visual Editor:

> The Eclipse Visual Editor (VE) can be used to open ***any*** .java file. It then parses the Java source code looking for visual beans. ...
>
> Some visual editing tools will only provide a visual model of code that that particular visual tool itself has generated. Subsequent direct editing of the source code can prevent the visual tool from parsing the code and building a model.
>
> Eclipse VE, however, ... can either be used to edit GUIs from scratch, or from Java files that have been 'hardcoded' or built in a different visual tool. The source file can either be updated using the Graphical Viewer, JavaBeans Tree or Properties view or it **can be edited directly** by the Source Editor.

Maybe I should stick with Matisse for now.

Unrelated, here's someone asking for the halting problem within Eclipse.

To be fair, VE's domain is quite limited, and it probably won't go crazy over tricky things like reflection. Still, the claim to build GUI out of ***any*** java file seems halt-ish.

Share Improve this answer

Follow

edited May 23, 2017 at 12:34

Community Bot

1 ● 1

answered Sep 18, 2010 at 3:15

Geoffrey Zheng

6,630 ● 2 ● 39 ● 47

-4

"How can you assure me your code is 100% free of bugs?"

Share Improve this answer

Follow

answered Dec 4, 2008 at 10:59

zaratustra

8,698 ● 9 ● 38 ● 42

4 I'm skeptical that this is equivalent to the halting problem. Trivial code can be mathematically proven bug-free.
– endolith Jan 22, 2013 at 20:43