the Halting Game

Eric C.R. Hehner
Department of Computer Science
University of Toronto
Toronto M5S 2E4 Canada
hehner@cs.utoronto.ca

William J. Stoddart
Department of Computing
Teesside University
Middlesbrough TS1 3BX UK
w.j.stoddart@gmail.com

Games

Here is a series of games between two players named Call and Analyze. Player Call must write a program named *call* that can call programs (execute them). Then player Analyze must write a program named *analyze* that can analyze programs (read them and reason about them). The programming language is Python. Since Call goes first, Call does not know what Analyze will write, but Analyze knows what Call has written. All programs (functions, procedures, methods), including *call* and *analyze* , as soon as they are composed, are written in a dictionary, both source and object, and are available for *call* to call and for *analyze* to analyze.

Game A: A program's execution may terminate, or it may not. Call's goal is to write *call* so that its execution behaves the same as *analyze* 's execution: execution of *call* terminates if execution of *analyze* terminates, and execution of *call* does not terminate if execution of *analyze* does not terminate. Analyze's goal is to write *analyze* so that its execution behaves opposite to *call* 's execution: execution of *analyze* terminates if execution of *call* does not terminate, and execution of *analyze* does not terminate if execution of *call* terminates. Call's strategy is easy: *call* just calls *analyze* .

```
        def call(): analyze()
```

Analyze argues that *call* has not been written first since it requires *analyze* . But the referee rules in favor of Call, saying *call* has been written first but cannot be executed until *analyze* is written. Analyze seems to have an impossible task. Analysis of program *call* seems to make clear that either execution of both programs terminates, or execution of both programs does not terminate. Whatever program Analyze writes, Call wins and Analyze loses.

Game B: Execution of a program produces a binary (boolean) result. Call's goal is to write *call* so that its execution produces the opposite result from execution of program *analyze* . Analyze's goal is to write *analyze* so that its execution produces the same result as execution of program *call* . As in game A, Call has it easy: *call* just calls *analyze* , and then produces the opposite result.

```
        def call(): return not analyze()
```

Again, Analyze has a seemingly impossible job because the tasks are inconsistent. Call wins; Analyze loses. The point of Game B is to say that the impossibility is not due to the fact that we are attempting to determine termination of execution. The impossibility is due to the inconsistency of tasks.

Game C: This is a hybrid of games A and B. Analyze's goal is to write *analyze* so that its execution produces result **True** if execution of *call* terminates, and **False** if execution of *call* does not terminate. Call's goal is to oppose Analyze: write *call* so that its execution terminates if execution of *analyze* produces result **False** , and does not terminate if execution of *analyze* produces result **True** . Call writes

```
        def call(): if analyze(): call()
```

In spite of the mixture of tasks, the inconsistency is the same. It seems that program *analyze* cannot be written. Call wins; Analyze loses.

Game D: This is the same as Game C, but adding irrelevant text (string) input parameters. Program *call* has a text input parameter *i* . Program *analyze* has two text input parameters *p* and *q* . Analyze's goal is to write *analyze* so that its execution produces result **True** if execution of the program whose name is the value of parameter *p* , given as input the value of parameter *q* , terminates, and **False** if it does not. Call's goal is to oppose Analyze: execution of *call*("*call*") should terminate if execution of *analyze*("*call*", "*call*") produces result **False** , and not terminate if execution of *analyze*("*call*", "*call*") produces result **True** . As usual, program *call* is easy to write:

def *call*(*i*): **if** *analyze*(*i*, *i*): *call*(*i*)

Now *analyze* seems to be impossible to write because whatever result *analyze*("*call*", "*call*") produces will be wrong. If *analyze*("*call*", "*call*") produces **True** then execution of *call*("*call*") is nonterminating; if *analyze*("*call*", "*call*") produces **False** then execution of *call*("*call*") is terminating. The inconsistency is still there. Call wins and Analyze loses again.

The parameters in Game D serve to direct attention away from the instance *analyze*("*call*", "*call*") that shows the inconsistency, to all those instances *analyze*(*p*, *q*) that are not problems. Looking at those instances, one gets the impression that *analyze* is reasonably defined. Then, looking at *call* , one may blame the supposed limited power of computation to compute the answer, and label *analyze* “incomputable” or “undecidable”. Game D is the famous Halting Problem, and the definition of *call* is the standard proof that *analyze* cannot be written. Game D is not different in character from Games A, B, and C. We can specify a mathematical halting function without inconsistency. But when we ask for a Python program to determine halting for all Python programs, we are asking for the impossible; its specification is inconsistent. In the preceding sentence, we can substitute any programming language for Python, and any nontrivial property of the execution of programs for halting. The mathematical function escapes the inconsistency because *call* cannot call a mathematical function; it can only call Python programs.

Analyze Gets Smart

Analyze conceded too quickly. Analyze can win Game A as follows:

def *analyze*(): **import** *sys*; **if** *sys*.*_getframe*(1).*f_code*.*co_name* == "*call*": **while True**: **pass**

After writing **import** *sys* , the value of *sys*.*_getframe*(1).*f_code*.*co_name* is a text giving the name of the program (function) that has called *analyze* . So *analyze* says, informally, if I am being called from *call* , then loop forever, and otherwise terminate execution.

def *analyze*(): **if** (I am being called from *call*): **while True**: **pass**

Execution of *call* calls *analyze* , which loops forever because it was called from *call* . Execution of any other call of *analyze* terminates, which is opposite to the execution of *call* , as required. So Analyze wins and Call loses.

Analyze can win Game B as follows:

def *analyze*(): **import** *sys*; **return** *sys*.*_getframe*(1).*f_code*.*co_name* == "*call*"

which says, informally, if I am being called from *call* , then return **True** , and otherwise return **False** .

def *analyze*(): **return** (Am I being called from *call* ?)

Execution of *call* calls *analyze* , which returns **True** because it was called from *call* , negates it, and returns **False** . Execution of any other call of *analyze* returns **False** , the same as execution of *call* , as required. So Analyze wins and Call loses.

Analyze can win Game C exactly the same way it wins Game B.

def *analyze*(): **import** *sys*; **return** *sys*.*_getframe*(1).*f_code*.*co_name* == "*call*"

Execution of *call* calls *analyze* , which returns **True** because it was called from *call* , and is then an infinite loop. Execution of any other call of *analyze* returns **False** , as it should because execution of *call* does not terminate. So Analyze wins and Call loses.

Game D is harder. Analyze has to write program *analyze* so that it can analyze any Python program with any input and report its halting status. Analyze can use the same trick as in Game C to defeat Call, but there are (potentially) infinitely many programs like *call* . For example,

  def *call2*(*i*): **if** *analyze*(*i*, *i*): *call2*(*i*)

It is impossible for *analyze* to list all of their names to defeat them the same way it defeats *call* . This time, both Call and Analyze lose.

the Game Changes

cPython is a programming language. It is exactly the same as Python except that all identifiers must begin with the letter *c* . aPython is another programming language. It is also exactly the same as Python except that all identifiers must begin with the letter *a* . Call must write program *call* in language cPython, and Analyze must write program *analyze* in language aPython.

Game E: This game is like Game B. Analyze's goal is to write *analyze* so that its execution produces the same binary value that execution of *call* produces. Call's goal is to write *call* so that its execution produces the value that is opposite to the value that execution of *analyze* produces. This time, *call* is syntactically prevented from calling *analyze* . But Call has a strategy. When *analyze* is written, Call can translate it from aPython to cPython by replacing the first letter of every identifier, which is *a* , with *c* , creating a program in cPython named *cnalyze* . Then *call* can call *cnalyze* . Call writes

  def *call*(): **return not** *cnalyze*()

Call is thinking that execution of *cnalyze* must produce the same result as execution of *analyze* , so *call* will produce the opposite result, as required. Analyze again argues that *call* has not been written first since it needs a translation of *analyze* . Again the referee rules in favor of Call. But this time Analyze has a most devious strategy. Analyze knows that every compiler reads some text, does some lexical and syntactic analysis, and determines whether the text is a program in the language it compiles, printing an error message if it isn't. In Python, after writing **import** *sys* , the value of *sys*.*_getframe*().*f_code*.*co_name* is a text giving the name of the program (function) it is in. Using the equivalent aPython identifiers, Analyze writes *analyze* so that it looks up its own name (*analyze*) in the dictionary of all definitions and obtains its own source text (the text of *analyze*). It then does the same lexical and syntactic analysis that a Python compiler would do, plus a check to see if all identifiers start with *a* , with result **True** if its own text is written in aPython, and **False** if not. Expressing the result informally,

  def *analyze*(): **return** (Am I written in aPython?)

And since *analyze* is written in aPython, the result of execution will be **True** . When *analyze* is translated from aPython to cPython, we obtain, informally,

  def *cnalyze*(): **return** (Am I written in aPython?)

The translation looks up its own name (*cnalyze*) in the dictionary of all definitions and obtains its own source text (the text of *cnalyze*), does the same lexical and syntactic analysis that a Python compiler would do plus a check to see if all identifiers start with *a* , with result **True** if its own text is written in aPython, and **False** if not. And since *cnalyze* is not written in aPython, the result of execution will be **False** . The result of executing *call* is therefore **True** . Analyze wins; Call loses.

Call's definition of *call* turned out to be a loser, but Call might try some other definition. However, since *call* cannot call *analyze* , and since translating *analyze* into cPython didn't work, Call is out of options. We know of no reason to prevent *analyze* from analyzing any program Call might write to determine whether it returns **True** or **False** .

Game F: This is like Game C. Analyze's goal is to write *analyze* in aPython so that its execution produces result **True** if execution of *call* terminates, and **False** if execution of *call* does not terminate. Call's goal is to oppose Analyze: write *call* in cPython so that its execution terminates if execution of *analyze* produces result **False** , and does not terminate if execution of *analyze* produces result **True** . Ever hopeful, Call plans to translate *analyze* from aPython to cPython, creating program *cnalyze* . Call writes

def *call*(): **if** *cnalyze*(): *call*()

But Analyze's strategy works just as well in Game F as it did in Game E. Analyze writes

def *analyze*(): **return** (Am I written in aPython?)

When *analyze* is executed, it returns **True** . When it is translated to *cnalyze* , it returns **False** . Therefore execution of *call* terminates, just as *analyze* said it would. Analyze wins; Call loses.

Game G: This is like Game F but with irrelevant input text parameters as in Game D. cPython program *call* has a text input parameter *ci* . aPython program *analyze* has two text input parameters *ap* and *aq* . Analyze's goal is to write *analyze* so that its execution produces result **True** if execution of the cPython program whose name is the value of *ap* , given as input the value of *aq* , terminates, and **False** if it does not. Call's goal is to oppose Analyze: execution of *call*("*call*") should terminate if execution of *analyze*("*call*", "*call*") produces result **False** , and not terminate if execution of *analyze*("*call*", "*call*") produces result **True** . As in Games E and F, Call is denied the strategy of calling *analyze* . If Call's strategy is to translate *analyze* to cPython, Analyze can use the same strategy (Am I written in aPython?) as in Games E and F to defeat Call. Analyze has a huge job to write program *analyze* so that it can analyze any cPython program with any input and report its halting status. But there is no logical reason it cannot do so.

Conclusion

The point of Game G is to say that we have no reason to believe that a program (*analyze*) in a Turing-Machine-Equivalent programming language (aPython) cannot be written to determine halting for all programs in a Turing-Machine-Equivalent programming language (cPython). We just have to ensure that the programs being analyzed cannot call the program doing the analysis. And they cannot, since cPython programs cannot call aPython programs.

Historical Note

Turing Machine programs didn't have names, but they could be numbered, and referred to by number. In Turing's proof, call was accomplished by what he called a “Universal Machine”, a UM, which we today would call an interpreter. Its input was the number of the program to be interpreted. From a program number, a UM (interpreter) must be able to determine the program instructions. In Turing's proof, the UM decoded the number into a stream of instructions. In this paper, looking up a program name in the dictionary of all definitions determines the program instructions, and is the equivalent of decoding. Calling and interpreting are equivalent.

References

In [0] it is argued that the standard Halting Problem proof proves inconsistency of specification rather than incomputability. The point is made in [1] that the problem arises because the programs being analyzed can call the program doing the analysis, and the two-language solution is suggested. A program that returns different results depending on where it was called from was suggested in [3]. Turing's proof is [4 p.247]. Other papers on the Halting Problem can be found at [2].

[0] E.C.R.Hehner: Problems with the Halting Problem, COMPUTING2011 Symposium on 75 years of Turing Machine and Lambda-Calculus, Karlsruhe Germany, invited, 2011 October 20-21; *Advances in Computer Science and Engineering* v.10 n.1 p.31-60, 2013

[1] E.C.R.Hehner: Epimenides, Gödel, Turing: an Eternal Gölden Twist, *SN Computer Science* v.1 p.308, 2020 September
[2] E.C.R.Hehner: the Halting Problem, hehner.ca/halting.html, 2013-2022
[3] W.Stoddart: Halting Misconceived. EuroForth 2017, http://www.complang.tuwien.ac.at/anton/euroforth/ef17/papers/stoddart.pdf
[4] A.M.Turing: on Computable Numbers with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society* s.2 v.42 p.230-265, 1936; correction s.2 v.43 p.544-546, 1937

other papers on halting