the Halting Game

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

William J. Stoddart  
Department of Computing  
Teesside University  
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 \texttt{call} that can call programs (execute them). Then player Analyze must write a program named \texttt{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 \texttt{call} and \texttt{analyze}, as soon as they are composed, are written in a dictionary, both source and object, and are available for \texttt{call} to call and for \texttt{analyze} to analyze.

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

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

Analyse argues that \texttt{call} has not been written first since it requires \texttt{analyze}. But the referee rules in favor of Call, saying \texttt{call} has been written first but cannot be executed until \texttt{analyze} is written. Analyse's goal is to write \texttt{analyze} so that its execution behaves opposite to \texttt{call}'s execution: execution of \texttt{analyze} terminates if execution of \texttt{call} does not terminate, and execution of \texttt{analyze} does not terminate if execution of \texttt{call} terminates. This seems to be an impossible task. Analysis of program \texttt{call} seems to make clear that either execution of both programs terminates, or execution of both programs does not terminate. Whatever program Analyse writes, Call wins and Analyse loses.

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

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

Again, Analyse has a seemingly impossible job because the tasks are inconsistent. Call wins; Analyse 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. Analyse's goal is to write \texttt{analyze} so that its execution produces result \texttt{True} if execution of \texttt{call} terminates, and \texttt{False} if execution of \texttt{call} does not terminate. Call's goal is to oppose Analyse: write \texttt{call} so that its execution terminates if execution of \texttt{analyze} produces result \texttt{False}, and does not terminate if execution of \texttt{analyze} produces result \texttt{True}. Call writes

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

In spite of the mixture of tasks, the inconsistency is the same. It seems that program \texttt{analyze} cannot be written. Call wins; Analyse loses.

Game D: This is the same as Game C, but adding irrelevant text (string) input parameters. Program \texttt{call} has a text input parameter \texttt{i}. Program \texttt{analyze} has two text input parameters \texttt{p} and \texttt{q}. Analyse'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:

```python
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 the problem instance, 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. It 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 programs.

**Analyze Gets Smart**

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

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

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.

```python
def analyze(): import sys; if (I am being called from call): while True:
    ...
```

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 Games B and C as follows:

```python
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.

```python
def analyze(): import sys; 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.

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. There are (potentially) infinitely many programs like call. For example,

```python
def call2(i): if analyze(i, i): call2(i)
```

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

PPython is a programming language. It is exactly the same as Python except that all identifiers must begin with the letter \( P \). QPython is another programming language. It is also exactly the same as Python except that all identifiers must begin with the letter \( Q \). Call must write program \( \text{Pcall} \) in language PPython, and Analyze must write program \( \text{Qanalyze} \) in language QPython.

Game E: This game is like Game B. Analyze's goal is to write \( \text{Qanalyze} \) so that its execution produces the same binary value that execution of \( \text{Pcall} \) produces. Call's goal is to write \( \text{Pcall} \) so that its execution produces the value that is opposite to the value that execution of \( \text{Qanalyze} \) produces. This time, \( \text{Pcall} \) is syntactically prevented from calling \( \text{Qanalyze} \). But Call has a strategy. When \( \text{Qanalyze} \) is written, Call can translate it from QPython to PPython by replacing the first letter of every identifier, which is \( Q \), with \( P \), creating a program in PPython named \( \text{Panalyze} \). Then \( \text{Pcall} \) can call \( \text{Panalyze} \). Call writes

\[
\text{def Pcall(): return not Panalyze()}
\]

Call is thinking that execution of \( \text{Panalyze} \) must produce the same result as execution of \( \text{Qanalyze} \), so \( \text{Pcall} \) will produce the opposite result, as required. Analyze again argues that \( \text{Pcall} \) has not been written first since it needs a translation of \( \text{Qanalyze} \). 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 \( \text{import sys} \), the value of \( \text{sys._getframe().f_code.co_name} \) is a text giving the name of the program (function) it is in. Using the equivalent QPython identifiers, Analyze writes \( \text{Qanalyze} \) so that it looks up its own name (\( \text{Qanalyze} \)) in the dictionary of all definitions and obtains its own source text (the text of \( \text{Qanalyze} \)). 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 \( Q \), with result \( \text{True} \) if its own text is written in QPython, and \( \text{False} \) if not. Expressing the result informally,

\[
\text{def Qanalyze(): return (Am I written in QPython?)}
\]

And since \( \text{Qanalyze} \) is written in QPython, the result of execution will be \( \text{True} \). When \( \text{Qanalyze} \) is translated from QPython to PPython, we obtain, informally,

\[
\text{def Panalyze(): return (Am I written in QPython?)}
\]

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

Call's definition of \( \text{Pcall} \) turned out to be a loser, but Call might try some other definition. However, since \( \text{Pcall} \) cannot call \( \text{Qanalyze} \), and since translating \( \text{Qanalyze} \) into PPython didn't work, Call is out of options. Any program Call might write can be analyzed by \( \text{Qanalyze} \) to determine whether its execution terminates.

Game F: This is like Game C. Analyze's goal is to write \( \text{Qanalyze} \) in QPython so that its execution produces result \( \text{True} \) if execution of \( \text{Pcall} \) terminates, and \( \text{False} \) if execution of \( \text{Pcall} \) does not terminate. Call's goal is to oppose Analyze: write \( \text{Pcall} \) in PPython so that its execution terminates if execution of \( \text{Qanalyze} \) produces result \( \text{False} \), and does not terminate if execution of \( \text{Qanalyze} \) produces result \( \text{True} \). Ever hopeful, Call plans to translate \( \text{Qanalyze} \) from QPython to PPython, creating program \( \text{Panalyze} \). Call writes

\[
\text{def Pcall(): if Panalyze(): Pcall()}
\]

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

\[
\text{def Qanalyze(): return (Am I written in QPython?)}
\]

When \( \text{Qanalyze} \) is executed, it returns \( \text{True} \). When it is translated to \( \text{Panalyze} \), it returns \( \text{False} \). Therefore execution of \( \text{Pcall} \) terminates, just as \( \text{Qanalyze} \) said it would. Analyze wins; Call loses.
Game G: This is like Game F but with irrelevant input text parameters as in Game D. PPython program `Pcall` has a text input parameter `Pi`. QPython program `Qanalyze` has two text input parameters `Qp` and `Qq`. Analyze's goal is to write `Qanalyze` so that its execution produces result `True` if execution of the PPython program whose name is the value of `Qp`, given as input the value of `Qq`, terminates, and `False` if it does not. Call's goal is to oppose Analyze: execution of `Pcall("Pcall")` should terminate if execution of `Qanalyze("Pcall", "Pcall")` produces result `False`, and not terminate if execution of `Qanalyze("Pcall", "Pcall")` produces result `True`. As in Games E and F, Call is denied the strategy of calling `Qanalyze`. If Call's strategy is to translate `Qanalyze` to PPython, Analyze can use the same strategy (Am I written in QPython?) as in Games E and F to defeat Call. Analyze has a huge job to write program `Qanalyze` so that it can analyze any PPython 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 (`Qanalyze`) in a Turing-Machine-Equivalent programming language (QPython) cannot be written to determine halting for all programs in a Turing-Machine-Equivalent programming language (PPython). We just have to ensure that the programs being analyzed cannot call the program doing the analysis.

**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, program names are associated with programs by the dictionary. Calling and interpreting are equivalent.

*other papers on halting*