Impossibility to Check Whether a Program – That Always Halts – Is Correct

Two important problems. When a freshman student starts to learn programming, the first big issue is to make sure that the program is syntactically correct – so that it can be compiled – and that it stops. Syntactic correctness is checked by a compiler. As for stopping, we have already proven that, in general, it is not possible to algorithmically check whether the program will halt or not.

Once a student gets a program that is compilable and that eventually stops, the next task is to make sure that this program always produces correct results. It turns out that this problem of checking correctness is also, in general, not algorithmically solvable.

We will first show that this is true even for the simplest specifications – e.g., for programs that always return 0. It is easy to write a program that always returns 0, but if someone provides a program and claims that it always returns 0, we will not always be able to check whether this claim is correct.

Theorem 1. *No algorithm is possible that, given a program p that always halts, checks whether this program always returns 0, i.e., whether $\forall n\,(p(n) = 0)$.*

Proof. We will prove this result by reduction to a contradiction. Let us assume that such an algorithm exists, i.e., that there exist an algorithm *zeroChecker* such that:

- if the program p always halts and $p(n) = 0$ for all n, then

$$zeroChecker(p) = 1;$$

- and if the program p always halts and $p(n) \neq 0$ for some n, then

$$zeroCheker(p) = 0.$$

Comment. If the program p does not halt on some data, we do not care what *zeroChecker* returns when applied to such a program p: it may return whatever it wants, it may go into an infinite loop – does not matter.

To get a contradiction, we will show that, based on the *zeroChecker*, we can build a halt-checker – and we have already proven that halt-checkers do not exist. Indeed, suppose that we are given a program p and data d, and we want to check whether p halts on d. This halt-checking is known to be, in general, not

algorithmically possible. However, what *is* algorithmically possible is to check, for each natural number t, whether p halts on d during the first t seconds. To check that, we simply run the program p on data d for t seconds and see if it halted. We can easily automate this, so we have a program $c_{p,d}(t)$ that, given a natural number t, checks whether p halts on d on or before t seconds, i.e.:

- returns $c_{p,d}(t) = 1$ ("true") is the program p halted on d on or before t seconds, and
- returns $c_{p,d}(t) = 0$ ("false") is the program p did not halt on d during the first t seconds.

The program $c_{p,d}(t)$ produces some result for all t, i.e., it always halts.

- What does it mean that the program p halts on d? It means that it halts after some time t, i.e., that $c_{p,d}(t) = 1$ for some t.
- What does it mean that the program p does not halt on d? It means that no matter what natural number t we choose, the program p will not halt during the first t seconds, i.e.., that $c_{p,d}(t) = 0$ for all t.

So, the program p does not halt on d if and only if the auxiliary program $c_{p,d}(t)$ always returns 0. Since we have a zero-checker, we can check ths property: $zeroChecker(c_{p,d}) = 1$ if and only if $\forall t\,(c_{p,d}(t) = 0)$, i.e., if and only if p does not halt on d. Thus:

- if the program p does not halt on d, we get $zeroChecker(c_{p,d}) = 1$, and
- if the program p halts on d, we get $zeroChecker(c_{p,d}) = 0$.

So, if we apply negation to $zeroChecker(c_{p,d})$, we get ourselves a halt-checker:

- if the program p halts on d, then $!zeroChecker(c_{p,d}) = 1$, and
- if the program p does not halt on d, then $!zeroChecker(c_{p,d}) = 0$.

We know that halt-checker do not exist, so we get a contradiction. The only assumption that we made to come up with this contradiction was that there exists a zero-checker. Thus, this assumption is wrong, and zero-checkers are not possible. The theorem is proven.

What about programs computing non-zero functions? It turns out that the negative result about any other computable function can be reduced to the impossibility of zero-checker. Let us illustrate it on the example of square-checker – checking whether the given program p always correctly computes the value $p(n) = n^2$.

Theorem 2. *No algorithm is possible that, given a program p that always halts, checks whether this program always returns n^2, i.e., whether $\forall n\,(p(n) = n^2)$.*

Proof. We will prove that if such a square-checker exists, then we can construct a zero-checker – and we already know that zero-checkers are not possible. Indeed, let us assume that we have an algorithm $squareChecker(p)$ that, given a

program p that always halts, checked whether $\forall n\,(p(n) = n^2)$. Suppose that we have a program q that always halts and we want to check whether this program q always returns 0. To check this, we form the following auxiliary program that always returns $q(n) + n^2$:

```
public static int aux(int n)
  {return q(n) + n * n;}
```

The value $q(n) + n^2$ is always equal to n^2 if and only if the value $q(n)$ is always equal to 0.

Thus, the algorithm $squareChecker(q(n) + n^2)$ that applies $squareChecker$ to the above auxiliary program is a zero-checker. However, we have proven that zero-checkers do not exist. This contradiction shows that our assumption – that square-checkers are possible – leads to a contradiction. Thus, square-checkers are not possible. The theorem is proven.