Chapter 6
Forecasting and Unpredictability

While – depending on one's subjective optimism or pessimism often, sometimes or rarely – it is possible to predict the future, certain forecasting tasks, in particular, when it comes to self-reference, are provable unattainable, and will remain so forever. Why? Because some forecasting tasks would result in the following situation, frugally explained by Aaronson [2] *"Turing imagined that there was a special machine that could solve the Halting Problem. Then he showed how we could have this machine analyse itself, in such a way that it has to halt if it runs forever, and run forever if it halts. Like a hound that finally catches its tail and devours itself, the mythical machine vanishes in a fury of contradiction. (That's the sort of thing you don't say in a research paper.)"*

6.1 Reduction from Logical Incompleteness

Given two problems A and B. Let us say that if "a reduction from problem A (in)to problem B" exists (or "problem A is reducible to problem B") then the solution to problem B can be used to solve problem A. Indeed, one may think of B as some "oracle" or "subroutine" which can be used to solve A. Thereby, reduction from A into B is an algorithm for transforming problem A into another problem B. Therefore, when problem A is reducible to problem B, then a solution of problem A cannot be harder than a solution to problem B, since a solution to B provides a solution to A. Hence, a reduction from problem A to another problem B can be used to show that problem B is at least as difficult as problem A.

More specifically, reduction (aka "algorithmic translation") from some unsolvable problem A (in particular, the halting problem) to problem B means the demonstration that the problem B in question is unsolvable by showing that the unsolvable problem A (in particular, the halting problem) can be reduced to it: that is, by showing that if we could compute a solution to problem B in question, we could use this solution to get a computable method for solving the unsolvable problem A (in particular, the

K. Svozil, *Physical (A)Causality*, Fundamental Theories of Physics 192,
https://doi.org/10.1007/978-3-319-70815-7_6

halting problem) [435, Sect. 2.1, p. 34]. But there cannot exist such a computable method of solving A. Therefore problem B must be unsolvable as well.

In what follows we shall follow previous reviews of that subject [499, 516]; mostly in the context of classical mechanics. Thereby the standard method is a reduction from some form of recursion theoretic incompleteness (in particular, the halting problem) into some physical entity or decision problem. Here the term *reduction* also refers to the method to link physical undecidability by reducing it to logical undecidability. Logical undecidability, in turn, can be related to ancient antinomies – for instance "the liar:" already the Bible's Epistle to Titus 1:12, states that *"one of Crete's own prophets has said it: 'Cretans are always liars, evil brutes, lazy gluttons.' He has surely told the truth."* – as well as antinomies plaguing Cantor's naive set theory.

A typical example for this strategy is the embedding of a Turing machine, or any type of computer capable of universal computation, into a physical system. As a consequence, the physical system inherits any type of unsolvability derivable for universal computers such as the unsolvability of the halting problem: because the computer or recursive agent is embedded within that physical system, so are its behavioural patterns.

References [35, 119–121, 154–156, 197, 284, 302, 372, 497, 499, 573, 574]. contain concrete examples. The author used a similar reduction technique in the context of a universal ballistic computational model to argue that the *n-body problem* [171, 413, 563] may perform in an undecidable manner; that is, some observables may not be computable. Consequently, the associated series solutions [496, 560, 561] might not have computable rates of convergence; just like Chaitin's Ω [108, 129, 136], the halting probability for prefix-free algorithms on universal computers [109, 111].

Of course, at some point this method or metaphor becomes problematic, as universal computation requires the arbitrary allocation of time and – depending of the computational model – computational and/or memory space; that is, a potentially infinite totality. This is never achievable in realistic physical situations [79–81, 232, 233].

6.2 Determinism Does Not Imply Predictability

One immediate consequence of reduction is the fact that, at least for sufficiently complex systems allowing the implementation of Peano arithmetic or universal computation, determinism does not imply predictability [497, 499]. This may sound counterintuitive at first but is quite easy to understand in terms of the behaviour, the temporal evolution or phenomenology of a device or subsystem capable of universal computation.

Let us, for the sake of a more explicit (but not formal and in a rather algorithmic way) demonstration what could happen, consider a supposedly and hypothetically *universal predictor*. We shall, by a *proof by contradiction* show, that the assumption of such a universal predictor (and some "innocent" side constructions) yields a complete

contradiction. Therefore, if we require consistency, our only consolation – or rather our sole option – is to abandon the assumption of the existence of a universal predictor.

6.2.1 Unsolvability of the Halting Problem

The scheme of the proof by contradiction is as follows: the existence of a hypothetical halting algorithm capable of solving universal prediction will be *assumed.* More specifically, it will be (wrongly) assumed that a "universal predictor" exists which can forecast whether or not any particular program halts on any particular input. This could, for instance, be a subprogram of some suspicious super-duper macro library that takes the code of an arbitrary program as input and outputs 1 or 0, depending on whether or not the respective program halts. One may also think of it as a sort of oracle or black box analysing an arbitrary program in terms of its symbolic code and outputting one of two symbolic states, say, 1 or 0, referring to termination or nontermination of the input program, respectively.

On the basis of this *hypothetical halting algorithm* one constructs another *diagonalization program* as follows: on receiving some arbitrary *input program* code (including its input code) as input, the diagonalization program consults the *hypothetical halting algorithm* to find out whether or not this input program halts. Upon receiving the answer, it does the exact *opposite* consecutively: If the hypothetical halting algorithm decides that the input program *halts,* the diagonalization program does *not halt* (it may do so easily by entering an infinite loop). Alternatively, if the hypothetical halting algorithm decides that the input program does *not halt,* the diagonalization program will *halt* immediately.

The diagonalization program can be forced to execute a paradoxical task by receiving *its own program code* as input. This is so because, by considering the diagonalization program, the hypothetical halting algorithm steers the diagonalization program into *halting* if it discovers that it *does not halt;* conversely, the hypothetical halting algorithm steers the diagonalization program into *not halting* if it discovers that it *halts.*

The contradiction obtained in applying the diagonalization program to its own code proves that this program and, in particular, the hypothetical halting algorithm as the single and foremost nontrivial step in the execution, cannot exist. A slightly revised form of the proof using quantum diagonalization operators holds for quantum diagonalization [512], as quantum information could be in a fifty-fifty fixed-point halting state. Procedurally, in the absence of any fixed-point halting state, the aforementioned task might turn into a nonterminating alteration of oscillations between halting and nonhalting states [303].

6.2.2 *Determinism Does Not Imply Predictability*

A very general result about the incomputability of nontrivial functional properties is *Rice's theorem* (Cf. the Appendix Sect. A.5 on p. 174) stating that, given an algorithm, all functional properties (that is, some "nontrivial" input/output behavior which neither is true for every program, nor true for no program – that is, some programs show this behaviour, and others don't) of that algorithm are undecidable. Stated differently, given a program, there is no general algorithm predicting or determining whether the function it computes has or has not some property (which some programs have, and others do not have).

One proof is by reduction to the halting problem; that is, a proof by contradiction: we construct a decision problem about functional properties by overlaying it with a primary halting problem. A the primary halting problem will in general be undecidable, so will be the compounded decision problem about function properties.

Suppose (wrongly) that there exists a program predicting or determining whether or not, for any given program, the function it computes has or has not some particular property (which some programs have, and others do not have).

Then we construct another program which first solves the halting program from some other arbitrary but definite program, then clears the memory, and after that, in a third step, runs a program which has the property which we are interested in. Now we apply this new program to the predictor. Suppose the other arbitrary but definite program terminates, then the predictor could in principle predict that the new program satisfies the property.

Alas, if the other arbitrary but definite program does not halt (but for instance goes into an infinite loop), then our predictor will never be able to execute the two final steps of the new program – that is, clearing the memory and running the program with the property we are interested in. Therefore, predicting the functional property for the new three-step program constructed amounts to deciding the halting problem for the other arbitrary but definite program. This task is in general undecidable for arbitrary other but definite programs.

6.3 Quantitative Estimates in Terms of the Busy Beaver Function

More quantitatively one can interpret this unpredictability in terms of the busy beaver function [71, 125, 168, 426], also discussed in Appendix A.7, which can be defined as a sort of "worst case scenario" as follows: suppose one considers all programs (on a particular computer) up to length (in terms of the number of symbols) n. What is the *largest number* producible by such a program before halting? (Note that non-halting programs, possibly producing an infinite number, e.g., by a non-terminating loop, do not apply.) This number may be called the *busy beaver function* of n.

Consider a related question: what is the upper bound of running time – or, alternatively, recurrence time – of a program of length n bits before terminating or, alternatively, recurring? An answer to this question will explain just how long we have to wait for the most time-consuming program of length n bits to halt. That, of course, is a worst-case scenario. Many programs of length n bits will have halted long before the maximal halting time. We mention without proof [125, 128] that this bound can be represented by the busy beaver function.

Knowledge of the maximal halting time would solve the halting problem quantitatively because if the maximal halting time were known and bounded by any computable function of the program size of n bits, one would have to wait just a little longer than the maximal halting time to make sure that every program of length n – also this particular program, if it is destined for termination – has terminated. Otherwise, the program would run forever. Hence, because of the recursive unsolvability of the halting problem the maximal halting time cannot be a computable function. Indeed, for large values of n, the maximal halting time "explodes in a way which is unbounded by computability;" thereby growing faster than any computable function of n (such as the Ackermann function).

By reduction, upper bounds for the recurrence of any kind of physical behaviour can be obtained; for deterministic systems representable by n bits, the maximal recurrence time grows faster than any computable number of n. This bound from below for possible behaviours may be interpreted quite generally as a measure of the impossibility to predict and forecast such behaviours by algorithmic means.